Theoretical Computer Science

Electronic Notes in Theoretical Computer Science

[Home Page of ENTCS] [Volume/Issue List of ENTCS] [Author Index of ENTCS]

HOOTS II Second Workshop on Higher-Order Operational Techniques in Semantics

Stanford University, December 8-12, 1997

Guest Editors: Andrew Gordon, Andrew Pitts and Carolyn Talcott

[Table of Contents] of Volume 10

ELSEVIER
Mirror sites: www.europe | www.usa | www.japan

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (Leave Blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	1998	Final 30 Sep 97 - 30 Sep 98

4. TITLE AND SUBTITLE HOOTS II Second Workshop on Higher-Order Operational Techniques in Semantics (Proceedings)

5. FUNDING NUMBERS
G: N00014-98-1-0201

6. AUTHORS (Editors)
Andrew Gordon, Andrew Pitts, & Carolyn Talcott

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Stanford University

8. PERFORMING ORGANIZATION REPORT NUMBER
NA

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)
ONR

10. SPONSORING / MONITORING AGENCY REPORT NUMBER
NA

11. SUPPLEMENTARY NOTES The proceedings are published in the series Electronic Notes in Theoretical Computer Science as Volume 10 and are available at http://www.elsevier.nl/locate/entcs/volume10.html

12a. DISTRIBUTION / AVAILABILITY STATEMENT
Approved for public release; distribution unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT (Maximum 200 words)

This issue of ENTCS is an unrefereed conference record of talks presented at the Second Workshop on Higher Order Operational Techniques in Semantics held at Stanford University, December 8-11, 1997. The meeting was organised by A. Gordon, A. Pitts and C. Talcott with generous sponsorship from Harlequin Ltd, NSF and ONR. The study of operational techniques for higher-order languages has much research activity going on in distinct communities, including the concurrency, functional programming and type theory communities. The purpose of the workshop was to bring researchers from these communities together to discuss current trends in the theory of operational semantics, its application to higher-order languages and its connection with more established semantic techniques.

14. SUBJECT TERMS			15. NUMBER OF PAGES
Semantics, Higher Order			16. PRICE CODE
17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL

Theoretical Computer Science

Electronic Notes in Theoretical Computer Science

Table of Contents of Volume 10

(Preliminary Report)
Gilles Barthe , John Hatcliff and Peter Thiemann
[Abstract] [Full text] (PostScript 1093.3 Kb)

Adapting Big-Step Semantics to Small-Step Style: Coinductive Interpretations and "Higher-Order" Derivations
Husain Ibraheem and David A. Schmidt
[Abstract] [Full text] (PostScript 564.9 Kb)

An Operational Semantics Framework Supporting the Incremental Construction of Derivation Trees
Allen Stoughton
[Abstract] [Full text] (PostScript 468.5 Kb)

Computing with Contexts: A simple approach
Dave Sands
[Abstract] [Full text] (PostScript 618.6 Kb)

Flow Logic and Operational Semantics
Flemming Nielson and Hanne Riis Nielson
[Abstract] [Full text] (PostScript 562.5 Kb)

An Introduction to History Dependent Automata
Ugo Montanari and Marco Pistore
[Abstract] [Full text] (PostScript 778.1 Kb)

Can Actors and pi-Agents Live Together?
Ugo Montanari and Carolyn Talcott
[Abstract] [Full text] (PostScript 447.1 Kb)

Specification Diagrams for Actor Systems
Scott Smith
[Abstract] [Full text] (PostScript 724.2 Kb)

Mobile Ambients
(Extended Abstract)
Luca Cardelli and Andrew D. Gordon
[Abstract] [Full text] (PostScript 290.4 Kb)

Secure Implementation of Channel Abstractions
Martin Abadi, Cedric Fournet and Georges Gonthier
[Abstract] [Full text] (PostScript 257.7 Kb)

Program Units as Higher-Order Modules
Matthew Flatt and Matthias Felleisen
[Abstract] [Full text] (PostScript 749.9 Kb)

Typed Closure Conversion for Recursively-Defined Functions
(Extended Abstract)
Greg Morrisett and Robert Harper
[Abstract] [Full text] (PostScript 412.3 Kb)

A Type System For Object Initialization In the Java Bytecode Language
Steve Freund and John Mitchell
[Abstract] [Full text] (PostScript 387.5 Kb)

Preface and Abstracts of Volume 10

Preface Volume 10

Andrew Gordon, Andrew Pitts and Carolyn Talcott

Abstract

This issue of ENTCS is an unrefereed conference record of talks presented at the Second Workshop on Higher Order Operational Techniques in Semantics (HOOTS II) held at Stanford University, December 8-11, 1997. The meeting was organised by A. Gordon, A. Pitts and C. Talcott with generous sponsorship from Harlequin Ltd, NSF and ONR. The first HOOTS workshop was held October 28-30, 1995 as part of the University of Cambridge Isaac Newton Institute research programme on Semantics of Computation (July-Dec 1995).

The study of operational techniques for higher-order languages is now a thriving area, with much research activity going on world-wide. An important open problem is a theory of program equivalence for languages with higher-order features, including functions and objects. Techniques for defining and reasoning about equivalence and other properties of higher-order programs have emerged in distinct communities, including the concurrency, functional programming and type theory communities. The purpose of the HOOTS workshops was to bring researchers from these communities together to discuss current trends in the theory of operational semantics, its application to higher-order languages and its connection with more established semantic techniques.

Papers presented at HOOTS II covered a broad range of topics:

- techniques such as bisimulation and logical relations for reasoning about contextual equivalence
- alternative program relations such as operational subsumption, and evaluation rules for program contexts
- operational models including adaptation of big-step evaluation semantics to provide capabilities of small-step and denotational semantics forms, flow graphs, and history dependent automata
- higher-order programming calculi including: imperative call-by-need lambda calculus, action calculi, process calculi for reasoning about mobility and security, interaction of actors

and pi calculus agents
- approaches to program analysis and verification, including: logics for control flow analysis, monadic type systems, and diagramatic specification notation for actor systems;
- programming environment tools such a type systems for Java byte-code, and higher-order program units for modularity.

Programs and participants lists for HOOTS I and II and other information about HOOTS, past and future can be found here

[back to table of contents]

Similarity and Bisimilarity for Countable Non-Determinism and Higher-Order Functions

(Extended Abstract)

Soren Lassen and Corin Pitcher

Abstract

This paper investigates operationally-based theories of a simply-typed functional programming language with countable non-determinism. The theories are based upon lower, upper, and convex variants of applicative similarity and bisimilarity, and the main result presented here is that these relations are compatible. The differences between the relations are illustrated by simple examples, and their continuity properties are discussed. It is also shown that, in some cases, the addition of countable non-determinism to a programming language with finite non-determinism alters the theory of the language.

[Full text] (PostScript 631.2 Kb)

[back to table of contents]

Parametric Polymorphism and Operational Equivalence

(Preliminary Version)

Andy Pitts

Abstract

Studies of the mathematical properties of impredicatively polymorphic types have for the most part focused on the polymorphic lambda calculus of Girard-Reynolds, which is a calculus of total polymorphic functions. This paper considers polymorphic types from a functional programming perspective, where the partialness arising from the presence of fixpoint recursion complicates the nature of potentially infinite ('lazy') datatypes. An operationally-based approach to Reynolds'

notion of relational parametricity is developed for an extension of Plotkin's PCF with forall-types and lazy lists. The resulting logical relation is shown to be a useful tool for proving properties of polymorphic types up to a notion of operational equivalence based on Morris-style contextual equivalence.

[Full text] (PostScript 756.3 Kb)

[back to table of contents]

Operational Subsumption, an Ideal Model of Subtyping

Laurent Dami

Abstract

In a previous paper we have defined a semantic preorder called operational subsumption, which compares terms according to their error generation behaviour. Here we apply this abstract framework to a concrete language, namely the Abadi-Cardelli object calculus. Unlike most semantic studies of objects, which deal with typed equalities and therefore require explicitly typed languages, we start here from a untyped world. Type inference is introduced in a second step, together with an ideal model of types and subtyping. We show how this approach flexibly accommodates for several variants, and finally propose a novel semantic interpretation of structural subtyping as embedding-projection pairs.

[Full text] (PostScript 673.1 Kb)

[back to table of contents]

An Operational Understanding of Bisimulation from Open Maps

Glynn Winskel

Abstract

Models can be given to a range of programming languages combining concurrent and functional features in which presheaf categories are used as the semantic domains (instead of the more usual complete partial orders). Once this is done the languages inherit a notion of bisimulation from the ''open'' maps associated with the presheaf categories. However, although there are methodological and mathematical arguments for favouring semantics using presheaf categories---in particular, there is a "domain theory" based on presheaf categories which systematises bisimulation at higher-order---it is as yet far from a routine matter to read off an "operational characterisation"; by this I mean an equivalent coinductive definition of bisimulation between terms based on the operational semantics. I hope to illustrate the issues on a little process-passing language. This is joint work with Gian Luca Cattani.

[back to table of contents]

Premonoidal categories and flow graphs

Alan Jeffrey

Abstract

We give two presentations of the semantics of programs: a categorical semantics based on Power and Robinson's symmetric premonoidal categories and Joyal, Street and Verity's traced monoidal categories, and a graphical semantics based on mixed control flow and data flow graphs. We show how these semantics are related, and sketch how the 2-categorical versions could be used to give an operational semantics for programs. The semantics is similar to Hasegawa's presentation of Milner and Gardner's name-free action calculi.

[back to table of contents]

A Type-theoretic Description of Action Calculi

Philippa Gardner

Abstract

Action calculi, introduced by Milner, provide a framework for investigating models of interaction. This talk will focus on the connection between action calculi and known concepts arising from type theory. The aim of this work is to isolate what is distinctive about action calculi, and to investigate the potential of action calculi as an underlying framework for many kinds of computational behaviour.

The first part of the talk will introduce action calculi. In the second part, I'll give a type-theoretic account of action calculi, using the general binding operators of Aczel. I will discuss two extensions: higher-order action calculi which correspond to Moggi's commutative computational lambda-calculus, and linear action calculi which correspond to the linear type theories of Barber and Benton.

This talk is based on joint work with Andrew Barber, Masahito Hasegawa and Gordon Plotkin. If time, I will also describe current work arising from the connections described above.

[back to table of contents]

Correctness of Monadic State: An Imperative Call-by-Need Calculus

Zena Ariola and Amr Sabry

Abstract

The extension of Haskell with a built-in state monad combines mathematical elegance with operational efficiency:

- Semantically, at the source language level, constructs that act on the state are viewed as functions that pass an explicit store data structure around.
- Operationally, at the implementation level, constructs that act on the state are viewed as statements whose evaluation has the side-effect of updating the implicit global store in place.

There are several unproven conjectures that the two views are consistent. Recently, we have noted that the consistency of the two views is far from obvious: all it takes for the implementation to become unsound is one judiciously-placed beta-step in the optimization phase of the compiler. This discovery motivates the current paper in which we formalize and show the correctness of the implementation of monadic state. For the proof, we first design a typed call-by-need language that models the intermediate language of the compiler, together with a type-preserving compilation map. Second, we show that the compilation is semantics-preserving by proving that the compilation of every source axiom yields an observational equivalence of the target language. Because of the wide semantic gap between the source and target languages, we perform this last step using a number of intermediate languages. The imperative call-by-need lambda-calculus is of independent interest for reasoning about system-level Haskell code providing services such as memo-functions, generation of new names, etc, and is the starting point for reasoning about the space usage of Haskell programs.

[back to table of contents]

Monadic Type Systems: Pure Type Systems for Impure Settings

(Preliminary Report)

Gilles Barthe , John Hatcliff and Peter Thiemann

Abstract

Pure type systems and computational monads are two parameterized frameworks that have proved to be quite useful in both theoretical and practical applications. We join the foundational concepts of both of these to obtain monadic type systems. Essentially, monadic type systems inherit the parameterized higher-order type structure of pure type systems and the monadic term and type structure used to capture computational effects in the theory of computational monads. We demonstrate that monadic type systems nicely characterize previous work and suggest how they can support several new theoretical and practical applications.

A technical foundation for monadic type systems is laid by recasting and scaling up the main results from pure type systems (confluence, subject reduction, strong normalisation for particular classes of systems, etc.) and from operational presentations of computational monads (notions of operational equivalence based on applicative similarity, co-induction proof techniques).

We demonstrate the use of monadic type systems with case studies of several call-by-value and call-by-name systems. Our framework allows to capture the restriction to value polymorphism in the type structure and is flexible enough to accommodate extensions of the type system, e.g., with

higher-order polymorphism. The theoretical foundations make monadic type systems well-suited as a typed intermediate language for compilation and specialization of higher-order, strict and non-strict functional programs. The monadic structure guarantees sound compile-time optimizations and the parameterized type structure guarantees sufficient expressiveness.

[Full text] (PostScript 1093.3 Kb)

[back to table of contents]

Adapting Big-Step Semantics to Small-Step Style: Coinductive Interpretations and "Higher-Order" Derivations

Husain Ibraheem and David A. Schmidt

Abstract

We adapt Kahn-style ("big-step") natural semantics to take on desirable aspects of small-step and denotational semantics forms, more precisely: (i) the ability to express divergent computations; (ii) the ability to reason about the (length of a) computation of a derivation; and (iii) the ability to compute upon and reason about higher-order values. To accomplish these results, we extend the classical, inductive interpretation of natural semantics with coinduction mechanisms and use "negative" rules to express divergence. A simple reformatting of the syntax of derivations allows a simple description of the "length" of a derivation. Finally, the recoding of closure values into denotational-semantics-like functions lets one embed derivations within closures that embed within derivations; in this sense, the semantics becomes "higher order." Examples are given to support the definitional developments.

[Full text] (PostScript 564.9 Kb)

[back to table of contents]

An Operational Semantics Framework Supporting the Incremental Construction of Derivation Trees

Allen Stoughton

Abstract

We describe the current state of the design and implementation of Dops, a framework for Deterministic OPerational Semantics that will support the incremental construction of derivation trees, starting from term/input pairs. This process of derivation tree expansion may terminate with either a complete derivation tree, explaining why a term/input pair evaluates to a particular output, or with a blocked incomplete derivation tree, explaining why a term/input pair fails to evaluate to an output; or the process may go on forever, yielding, in the limit, an infinite incomplete derivation tree, explaining why a term/input pair fails to evaluate to an output.

The Dops metalanguage is a typed lambda calculus in which all expressions converge. Semantic rules are specified by lambda terms involving resumptions, which are used by a rule to consume the outputs of sub-evaluations and then resume the rule's work. A rule's type describes the number and kinds of sub-evaluations that the rule can initiate, and indicates whether the rule can block. The semantics of Dops is defined in an object language-independent manner as a small-step semantics on concrete derivation trees: trees involving resumptions. These concrete derivation trees can then be abstracted into ordinary derivation trees by forgetting the resumptions.

[Full text] (PostScript 468.5 Kb)

[back to table of contents]

Computing with Contexts: A simple approach

Dave Sands

Abstract

This article describes how the use of a higher-order syntax representation of contexts [due to A. Pitts] combines smoothly with higher-order syntax for evaluation rules, so that definitions can be extended to work over contexts. This provides "for free" -- without the development of any new language-specific context calculi - evaluation rules for contexts which commute with hole-filling. We have found this to be a useful technique for directly reasoning about operational equivalence. A small illustration is given based on a unique fixed-point induction principle for a notion of guarded context in a functional language.

[Full text] (PostScript 618.6 Kb)

[back to table of contents]

Flow Logic and Operational Semantics

Flemming Nielson and Hanne Riis Nielson

Abstract

Flow logic is a ''fast prototyping'' approach to program analysis that shows great promise of being able to deal with a wide variety of languages and calculi for computation. However, seemingly innocent choices in the flow logic as well as in the operational semantics may inhibit proving the analysis correct. Our main conclusion is that environment based semantics is more flexible than either substitution based semantics or semantics making use of structural congruences (like alpha-renaming).

[Full text] (PostScript 562.5 Kb)

[back to table of contents]

An Introduction to History Dependent Automata

Ugo Montanari and Marco Pistore

Abstract

Automata (or labeled transition systems) are widely used as operational models in the field of process description languages like CCS. There are however classes of formalisms that are not modelled adequately by the automata. This is the case, for instance, of the pi-calculus, an extension of CCS where channels can be used as values in the communications and new channels can be created dynamically. Due to the necessity to represent the creation of new channels, infinite automata are obtained in this case also for very simple agents and a non-standard definition of bisimulation is required.

In this paper we present an enhanced version of automata, called history dependent automata, that are adequate to represent the operational semantics of pi-calculus and of other history dependent formalisms. We also define a bisimulation equivalence on history dependent automata, that captures pi-calculus bisimulation.

[Full text] (PostScript 778.1 Kb)

[back to table of contents]

Can Actors and pi-Agents Live Together?

Ugo Montanari and Carolyn Talcott

Abstract

The syntax and semantics of actors and pi-agents is first defined separately, using a uniform, ''unbiased'' approach. New coordination primitives are then added to the union of the two calculi which allow actors and pi-agents to cooperate.

[Full text] (PostScript 447.1 Kb)

[back to table of contents]

Specification Diagrams for Actor Systems

Scott Smith

Abstract

Traditional approaches to specifying distributed systems include temporal logic specification (e.g. TLA), and process algebra specification (e.g. LOTOS). We propose here a new form of graphical

notation for specifying open distributed object systems. The primary design goal is to make a form of notation for defining message-passing behavior that is expressive, intuitively understandable, and that has a formal underlying semantics. We describe the language and its use through presentation of a series of example specifications. We also give an operationally-based interaction path semantics for specification diagrams.

[Full text] (PostScript 724.2 Kb)

[back to table of contents]

Mobile Ambients

(Extended Abstract)

Luca Cardelli and Andrew D. Gordon

Abstract

There are two distinct areas of work in mobility: "mobile computing", concerning computation that is carried out in mobile devices, and "mobile computation", concerning mobile code that moves between devices. These distinctions are destined to vanish. We aim to describe all aspects of mobility within a single framework that encompasses mobile agents, the ambients where agents interact and the mobility of the ambients themselves.

The main difficulty with mobile computation is not in mobility per se, but in the crossing of administrative domains. Mobile programs must be equipped to navigate a hierarchy of domains, at every step obtaining authorization to move further. Therefore, at the most fundamental level we need to capture notions of locations, of mobility and of authorization to move.

We identify "mobile ambients" as a fundamental abstraction that generalizes both dynamic agents and the static domains they must cross. From a formal point of view we develop a simple but computationally powerful calculus that directly embodies domains and mobility (and little else). The calculus forms the basis of a small-language/Java-library. We demonstrate the expressiveness of the approach by a series of examples, including showing how a notion such as "crossing a firewall" has a direct and analyzable interpretation.

[Full text] (PostScript 290.4 Kb)

[back to table of contents]

Secure Implementation of Channel Abstractions

Martin Abadi, Cedric Fournet and Georges Gonthier

Abstract

While cryptography is useful for distributed applications and fun even for application programmers, cryptographic manipulations by and large do not belong in application code. Ideally, application code should not be concerned with the details of key management, but should instead rely on abstractions and services that encapsulate cryptographic protocols. In recent years, several APIs (application program interfaces) for security have appeared, providing such abstractions and services. Although there are substantial differences among these APIs, they generally offer the promise of making application code more modular, simpler, and ultimately more robust.

In this talk we consider high-level abstractions that largely hide the difficulties of network security from applications. These high-level abstractions support the pleasing illusion that all application address spaces are on the same machine, and that a centralized operating system provides security for them. In reality, these address spaces could be spread across a network, and security could depend on several local operating systems and on cryptographic protocols across machines. Thus, the application code need not be concerned with the security implications of distribution.

[Full text] (PostScript 257.7 Kb)

[back to table of contents]

Program Units as Higher-Order Modules

Matthew Flatt and Matthias Felleisen

Abstract

We have designed a new module language called ''program units''. Units support separate compilation, independent module reuse, cyclic dependencies, hierarchical structuring, and dynamic linking. In this paper, we present untyped and typed models of units.

[Full text] (PostScript 749.9 Kb)

[back to table of contents]

Typed Closure Conversion for Recursively-Defined Functions

(Extended Abstract)

Greg Morrisett and Robert Harper

Abstract

Much recent work on the compilation of statically typed languages such as ML relies on the propagation of type information from source to object code in order to increase the reliability and maintainabilty of the compiler itself and to improve the efficiency and verifiability of generated code. To achieve this the program transformations performed by a compiler must be cast as type-preserving translations between typed intermediate languages. In earlier work with Minamide

we studied one important compiler transformation, closure conversion, for the case of pure simply-typed and polymorphic lambda-calculus. Here we extend the treatment of simply-typed closure conversion to account for recursively-defined functions such as are found in ML. We consider three main approaches, one based on a recursive code construct, one based on a self-referential data structure, and one based on recursive types. We discuss their relative advantages and disadvantages, and sketch correctness proofs for these transformations based on the method of logical relations.

[Full text] (PostScript 412.3 Kb)

[back to table of contents]

A Type System For Object Initialization In the Java Bytecode Language

Steve Freund and John Mitchell

Abstract

In the standard Java implementation, a Java language program is compiled to Java bytecode and this bytecode is then interpreted by the Java Virtual Machine. Since bytecode may be written by hand, or corrupted during network transmission, the Java Virtual Machine contains a bytecode verifier that performs a number of consistency checks before code is interpreted. As one-step towards a formal specification of the verifier, we describe a precise specification of a subset of the bytecode language dealing with object creation and initialization.

[Full text] (PostScript 387.5 Kb)

[back to table of contents]

Electronic Notes in Theoretical Computer Science 10 (1998)
URL: http://www.elsevier.nl/locate/entcs/volume10.html 21 pages

Similarity and Bisimilarity for Countable Non-Determinism and Higher-Order Functions (Extended Abstract)

Søren B. Lassen [1]

University of Cambridge Computer Laboratory,
Pembroke Street, Cambridge CB2 3QG, United Kingdom
Soeren.Lassen@cl.cam.ac.uk

Corin S. Pitcher [2]

Oxford University Computing Laboratory,
Wolfson Building, Parks Road, Oxford OX1 3QD, United Kingdom
Corin.Pitcher@comlab.ox.ac.uk

Abstract

This paper investigates operationally-based theories of a simply-typed functional programming language with countable non-determinism. The theories are based upon lower, upper, and convex variants of applicative similarity and bisimilarity, and the main result presented here is that these relations are compatible. The differences between the relations are illustrated by simple examples, and their continuity properties are discussed. It is also shown that, in some cases, the addition of countable non-determinism to a programming language with finite non-determinism alters the theory of the language.

Key words: lambda-calculus, applicative bisimilarity, countable non-determinism.

1 Introduction

Non-deterministic programs are used in the study of concurrent systems, to abstract from scheduling details, and in methodologies for program construction, where specifications are regarded as non-deterministic programs. In recent years, several non-deterministic higher-order languages have been proposed in the literature in these areas (see, e.g., [28,4]). Non-determinism is

[1] Supported by a grant from the Danish Natural Science Research Council and grant number GR/L38356 from the UK EPSRC.
[2] Partially supported by a UK EPSRC studentship.

also found as an integrated feature of the higher-order, operationally-based semantic meta-language of action semantics [19]. In this paper we use operational techniques to study the interaction between non-determinism and higher-order functions in an idealised, minimal programming language.

We investigate three variants, lower, upper, and convex, of applicative similarity and bisimilarity for a simply-typed functional programming language with countable non-determinism. This builds upon work by Abramsky, Howe, and Ong [1,11,12,21] for deterministic and finitely non-deterministic higher-order calculi.

The variants of the relations correspond to the different constructions on preorders that are used to characterise the lower, upper and convex powerdomains. Their definitions refer to an inductively defined may convergence relation between terms, and also a co-inductively defined may divergence predicate on terms, for the upper and convex variants. For each variant there is an applicative similarity preorder and an applicative bisimilarity equivalence relation, both defined by co-induction. In addition, the applicative similarity preorders determine mutual applicative similarity equivalence relations that do not coincide with applicative bisimilarity. The proliferation of preorders and equivalences reflects the conflicting requirements of different applications for semantic theories of non-determinism. This complexity is not apparent in the absence of non-determinism, because the nine relations defined here collapse to just two.

It is of fundamental importance to know whether the relations are compatible, i.e., are they preserved by the constructors of the language? We prove that this is the case for all of the relations, extending methods due to Howe and Ong that were previously restricted to finitely non-deterministic languages for the upper and convex variants. By the use of induction on the derivation of a must convergence judgement (the complement of the may divergence predicate) their methods are extended smoothly to a language with countable non-determinism.

Must convergence is defined inductively via a finite collection of infinitary rule schema, and so ordinal heights can be assigned to the derivation trees of must convergence judgements in the usual way. Such trees have heights strictly less than ω for finite non-determinism, and heights strictly less than the least non-recursive ordinal ω_1^{CK} for countable non-determinism. This allows us to prove unwinding theorems for fixed points terms with respect to must convergence: ω-unwinding in the case of finitely non-deterministic terms, and a more unusual ω_1^{CK}-unwinding for countably non-deterministic terms.

2 A Functional Language with Non-Determinism

The vehicle for the examples and results in this paper is a variant of the language of Moggi's computational lambda-calculus [17,7]. Within the computational lambda-calculus there is a distinction between values and computations

that is enforced by the type system through a type-constructor for computation types. There are mechanisms for creating and composing the programs of computation types.

The language is extended with an operator $?\mathbb{N}$ to choose any natural number. The new construct is the sole source of non-determinism in the language, and, because it is assigned an appropriate computation type, non-determinism is restricted to the computation types. This restriction is convenient because the mechanism for composing computations can be used to control when non-determinism is resolved—an alternative is to incorporate both call-by-name and call-by-value abstractions (see, e.g., [22]). In addition, although the examples presented here have analogues at function types, they are simpler at computation types.

The types of the language are:

$$\sigma, \tau ::= \mathsf{unit} \mid \mathsf{nat} \mid \sigma \to \tau \mid \mathsf{P}(\sigma)$$

The computation types are those of the form $\mathsf{P}(\sigma)$, and the remaining types are called deterministic types.

The terms of the language are:

$$\begin{aligned} L, M, N ::= {} & x \mid \star \mid n \mid \mathsf{uop}\,(M) \mid \mathsf{bop}\,(M, N) \mid \\ & \mathsf{if}\, L\, \mathsf{then}\, M\, \mathsf{else}\, N \mid \lambda x : \sigma.\, M \mid M\, N \mid \\ & [M] \mid \mathsf{let}\, x : \sigma \Leftarrow M\, \mathsf{in}\, N \mid \mathsf{fix}\, x : \sigma.\, M \mid ?\mathbb{N} \end{aligned}$$

where x ranges over a countably infinite set of variables, n ranges over $\mathbb{N}$, and uop and bop range over a suitable set of symbols representing, respectively, unary and binary primitive recursive functions, e.g., not, plus, leq. For the sake of economy, booleans are represented by natural numbers: 0 for false, and 1 for true. The primitive recursive functions are assumed to follow this representation, and are denoted, e.g., $(\!|\mathsf{not}|\!) : \mathbb{N} \to \mathbb{N}$, $(\!|\mathsf{plus}|\!), (\!|\mathsf{leq}|\!) : \mathbb{N}^2 \to \mathbb{N}$. Variable binding terms follow the usual conventions for scope, α-conversion, and type annotations to ensure uniqueness of typing. The notation $M[N/x]$ denotes the capture-free substitution of N for free occurrences of x in M. The canonical terms are:

$$K ::= \star \mid n \mid \lambda x : \sigma.\, M \mid [M]$$

The type assignment rules in figure 1 are based on those of the computational lambda-calculus. We follow the convention that environments are partial functions, and that $\Gamma, x : \sigma$ is only defined when x is not in the domain of Γ.

The sets of terms and canonical terms that are closed and well-typed are *Exp* and *Can* respectively. A term that is closed and well-typed is called a program. The set of programs of type σ is Exp_σ.

$$\Gamma \vdash x : \sigma \quad (\Gamma(x) = \sigma) \qquad \Gamma \vdash \star : \mathsf{unit} \qquad \Gamma \vdash n : \mathsf{nat} \quad (n \in \mathbb{N})$$

$$\frac{\Gamma \vdash M : \mathsf{nat}}{\Gamma \vdash \mathsf{uop}\,(M) : \mathsf{nat}} \qquad \frac{\Gamma \vdash M : \mathsf{nat} \qquad \Gamma \vdash N : \mathsf{nat}}{\Gamma \vdash \mathsf{bop}\,(M, N) : \mathsf{nat}}$$

$$\frac{\Gamma \vdash L : \mathsf{nat} \qquad \Gamma \vdash M : \sigma \qquad \Gamma \vdash N : \sigma}{\Gamma \vdash \mathsf{if}\, L \,\mathsf{then}\, M \,\mathsf{else}\, N : \sigma}$$

$$\frac{\Gamma, x : \sigma \vdash M : \tau}{\Gamma \vdash \lambda x : \sigma.\, M : \sigma \to \tau} \qquad \frac{\Gamma \vdash M : \sigma \to \tau \qquad \Gamma \vdash N : \sigma}{\Gamma \vdash M\, N : \tau}$$

$$\frac{\Gamma \vdash M : \sigma}{\Gamma \vdash [M] : \mathsf{P}(\sigma)} \qquad \frac{\Gamma \vdash M : \mathsf{P}(\sigma) \qquad \Gamma, x : \sigma \vdash N : \mathsf{P}(\tau)}{\Gamma \vdash \mathsf{let}\, x : \sigma \Leftarrow M \,\mathsf{in}\, N : \mathsf{P}(\tau)}$$

$$\frac{\Gamma, x : \mathsf{P}(\sigma) \vdash M : \mathsf{P}(\sigma)}{\Gamma \vdash \mathsf{fix}\, x : \mathsf{P}(\sigma).\, M : \mathsf{P}(\sigma)} \qquad \Gamma \vdash ?\mathbb{N} : \mathsf{P}(\mathsf{nat})$$

Fig. 1. Type Assignment

Many of the examples that we give do not depend on the existence of distinct canonical programs at a base type, and in such cases we use the unit type in preference to nat.

The operational semantics is presented as an inductively defined evaluation relation $\Downarrow^{\mathrm{may}}$, called may convergence, between programs and canonical closed terms. The rules are shown in figure 2. The may convergence relation is not a partial function because of the rule that allows $?\mathbb{N}$ to converge to any natural number.

In contrast to the situation for deterministic programs, the divergent (non-terminating) behaviour of a non-deterministic program is not determined by its convergent behaviour. Following [6,8,18] we define a may divergence predicate $\Uparrow^{\mathrm{may}}$ on programs by co-induction. The may divergence rules are given in figure 3. The symbol $(-)$ at the side of each rule is used to indicate that the may divergence predicate is the greatest fixed point of the monotone function determined by the rules. Note that there is some redundancy in the may divergence rules, because it can be shown that programs of deterministic types cannot diverge.

Examples 2.1 and 2.2 highlight properties of the programming language that are relevant in the sequel.

Example 2.1 *The construct for sequencing computations in the computational lambda-calculus provides an additional degree of control over the resolution of non-determinism. For example, a call-by-value abstraction is definable at computation types (where y is fresh for M):*

$$\lambda^v x : \sigma.\, M \stackrel{\mathrm{def}}{=} \lambda y : \mathsf{P}(\sigma).\, \mathsf{let}\, x : \sigma \Leftarrow y \,\mathsf{in}\, M$$

The call-by-value abstraction exhibits (weak) call-time choice, i.e., a non-

$$K \Downarrow^{\text{may}} K \quad (K \in Can)$$

$$\frac{M \Downarrow^{\text{may}} n}{\mathsf{uop}\,(M) \Downarrow^{\text{may}} (\!|\mathsf{uop}|\!)(n)} \qquad \frac{M \Downarrow^{\text{may}} n_1 \qquad N \Downarrow^{\text{may}} n_2}{\mathsf{bop}\,(M,N) \Downarrow^{\text{may}} (\!|\mathsf{bop}|\!)(n_1,n_2)}$$

$$\frac{L \Downarrow^{\text{may}} n+1 \qquad M \Downarrow^{\text{may}} K}{\mathsf{if}\,L\,\mathsf{then}\,M\,\mathsf{else}\,N \Downarrow^{\text{may}} K} \quad (n \in \mathbb{N}) \qquad \frac{L \Downarrow^{\text{may}} 0 \qquad N \Downarrow^{\text{may}} K}{\mathsf{if}\,L\,\mathsf{then}\,M\,\mathsf{else}\,N \Downarrow^{\text{may}} K}$$

$$\frac{M \Downarrow^{\text{may}} \lambda x.\, M_1 \qquad M_1[N/x] \Downarrow^{\text{may}} K}{M\,N \Downarrow^{\text{may}} K}$$

$$\frac{M \Downarrow^{\text{may}} [M_1] \qquad N[M_1/x] \Downarrow^{\text{may}} K}{\mathsf{let}\,x \Leftarrow M\,\mathsf{in}\,N \Downarrow^{\text{may}} K}$$

$$\frac{M[\mathsf{fix}\,x.\,M/x] \Downarrow^{\text{may}} K}{\mathsf{fix}\,x.\,M \Downarrow^{\text{may}} K} \qquad ?\mathbb{N} \Downarrow^{\text{may}} [n] \quad (n \in \mathbb{N})$$

Fig. 2. May Convergence

deterministic program of type $\mathsf{P}(\sigma)$ *is resolved to program of type* σ *at the time that it is passed as an argument.*

Example 2.2 *Recursion is only available at computation types, but given a term* $\Gamma, f : \sigma \to \mathsf{P}(\tau) \vdash M : \sigma \to \mathsf{P}(\tau)$ *the following acts as a fixed point (where* g *and* x *are fresh for* M*):*

$$\Gamma \vdash \lambda x.\,\mathsf{let}\,f \Leftarrow \mathsf{fix}\,g.\,[\lambda x.\,\mathsf{let}\,f \Leftarrow g\,\mathsf{in}\,M\,x]\,\mathsf{in}\,M\,x : \sigma \to \mathsf{P}(\tau)$$

Non-determinism is often introduced via a binary operator, binary erratic choice. It can be defined in the programming language in terms of $?\mathbb{N}$.

Example 2.3 *The binary erratic choice of programs* M *and* N *of the same computation type is defined to be (where* y *is fresh for* M *and* N*):*

$$(M\,\mathsf{or}\,N) \stackrel{\text{def}}{=} \mathsf{let}\,y \Leftarrow ?\mathbb{N}\,\mathsf{in}\,\mathsf{if}\,y\,\mathsf{then}\,M\,\mathsf{else}\,N$$

Non-determinism is informally classified by the cardinalities of the sets of convergent behaviours of programs that cannot diverge (see section 6 also). For example, the binary erratic choice of deterministic terms is said to be finitely non-deterministic, whereas $?\mathbb{N}$ is said to be countably non-deterministic. König's lemma ensures that recursion does not provide a route from finite to countable non-determinism.

Example 2.4 *The program below can converge to any natural number, but may also diverge:*

$$\vdash \mathsf{fix}\,x.\,([0]\,\mathsf{or}\,\mathsf{let}\,y \Leftarrow x\,\mathsf{in}\,[\mathsf{plus}\,(y,1)]) : \mathsf{P}(\mathsf{nat})$$

It cannot be distinguished from $?\mathbb{N}$ *by equivalences that ignore divergent behaviour.*

$$\frac{M \Uparrow^{\text{may}}}{\mathsf{uop}\,(M) \Uparrow^{\text{may}}}- \qquad \frac{M \Uparrow^{\text{may}}}{\mathsf{bop}\,(M,N) \Uparrow^{\text{may}}}- \qquad \frac{N \Uparrow^{\text{may}}}{\mathsf{bop}\,(M,N) \Uparrow^{\text{may}}}-$$

$$\frac{L \Uparrow^{\text{may}}}{\mathsf{if}\,L\,\mathsf{then}\,M\,\mathsf{else}\,N \Uparrow^{\text{may}}}-$$

$$\frac{M \Uparrow^{\text{may}}}{\mathsf{if}\,L\,\mathsf{then}\,M\,\mathsf{else}\,N \Uparrow^{\text{may}}}- \quad (L \Downarrow^{\text{may}} n+1 \text{ and } n \in \mathbb{N})$$

$$\frac{N \Uparrow^{\text{may}}}{\mathsf{if}\,L\,\mathsf{then}\,M\,\mathsf{else}\,N \Uparrow^{\text{may}}}- \quad (L \Downarrow^{\text{may}} 0)$$

$$\frac{M \Uparrow^{\text{may}}}{M\,N \Uparrow^{\text{may}}}- \qquad \frac{M_1[N/x] \Uparrow^{\text{may}}}{M\,N \Uparrow^{\text{may}}}- \quad (M \Downarrow^{\text{may}} \lambda x.\, M_1)$$

$$\frac{M \Uparrow^{\text{may}}}{\mathsf{let}\,x \Leftarrow M\,\mathsf{in}\,N \Uparrow^{\text{may}}}- \qquad \frac{N[M_1/x] \Uparrow^{\text{may}}}{\mathsf{let}\,x \Leftarrow M\,\mathsf{in}\,N \Uparrow^{\text{may}}}- \quad (M \Downarrow^{\text{may}} [M_1])$$

$$\frac{M[\mathsf{fix}\,x.\,M/x] \Uparrow^{\text{may}}}{\mathsf{fix}\,x.\,M \Uparrow^{\text{may}}}-$$

Fig. 3. May Divergence

3 Similarity and Bisimilarity

Abramsky [1] develops a notion of applicative similarity for the untyped lazy lambda-calculus, building upon earlier work of Park and Milner [16] in the context of process calculi. The preorders and equivalences described in this section are based upon Abramsky's work and subsequent extensions to non-deterministic functional languages by Howe and Ong [12,21].

There are two fundamental differences between the deterministic and non-deterministic settings: applicative bisimilarity is not the same as mutual applicative similarity, and there are different ways of ordering non-deterministic programs that correspond to the constructions on preorders used to characterise the lower, upper, and convex powerdomains (see, e.g., [10,27]). This leads to nine distinct variations of applicative similarity and bisimilarity for non-deterministic programs, which collapse to just two relations on deterministic programs.

For the sake of brevity, "applicative" is implicit when similarity or bisimilarity are used in the sequel. The reader is also cautioned that terminology for (what we call) similarity or bisimilarity differs amongst authors. We use the following conventions: simulations and bisimulations are post-fixed points of a function; similarity and bisimilarity are the greatest simulations and bisimulations respectively; the prefix "bi" refers to a function on relations with a symmetric image; mutual similarity is the greatest symmetric relation contained in similarity.

The variants of similarity and bisimilarity are defined in terms of two functions of binary relations on programs. For a binary relation $\mathcal{R}$ on programs, we define binary relations on programs: $\langle\mathcal{R}\rangle_{\mathrm{LS}}$ and $\langle\mathcal{R}\rangle_{\mathrm{US}}$. The subscripts abbreviate lower similarity and upper similarity.

Definition 3.1 Let $\mathcal{R}$ be a binary relation on programs. The binary relations $\langle\mathcal{R}\rangle_{\mathrm{LS}}$ and $\langle\mathcal{R}\rangle_{\mathrm{US}}$ are defined by:

(i) $M, N \in Exp_\sigma$ are related by $\langle\mathcal{R}\rangle_{\mathrm{LS}}$ if:
  (a) $\sigma = \mathsf{unit}$; or
  (b) $\sigma = \mathsf{nat}$ and $\exists n \in \mathbb{N}.\ M \Downarrow^{\mathrm{may}} n \ \wedge\ N \Downarrow^{\mathrm{may}} n$; or
  (c) $\sigma = \tau_1 \to \tau_2$ and $\forall L \in Exp_{\tau_1}.\ (M\,L)\,\mathcal{R}\,(N\,L)$; or
  (d) $\sigma = \mathsf{P}(\tau)$ and $\forall M_1.M \Downarrow^{\mathrm{may}} [M_1] \Longrightarrow (\exists N_1.N \Downarrow^{\mathrm{may}} [N_1] \wedge M_1\,\mathcal{R}\,N_1)$.

(ii) $M, N \in Exp_\sigma$ are related by $\langle\mathcal{R}\rangle_{\mathrm{US}}$ if:
  (a) $\sigma = \mathsf{unit}$; or
  (b) $\sigma = \mathsf{nat}$ and $\exists n \in \mathbb{N}.\ M \Downarrow^{\mathrm{may}} n \ \wedge\ N \Downarrow^{\mathrm{may}} n$; or
  (c) $\sigma = \tau_1 \to \tau_2$ and $\forall L \in Exp_{\tau_1}.\ (M\,L)\,\mathcal{R}\,(N\,L)$; or
  (d) $\sigma = \mathsf{P}(\tau)$ and $\neg(M \Uparrow^{\mathrm{may}}) \Longrightarrow$
  $(\neg(N \Uparrow^{\mathrm{may}}) \wedge\ \forall N_1.N \Downarrow^{\mathrm{may}} [N_1] \Longrightarrow (\exists M_1.M \Downarrow^{\mathrm{may}} [M_1] \wedge M_1\,\mathcal{R}\,N_1))$.

The functions $\langle\cdot\rangle_{\mathrm{LS}}$ and $\langle\cdot\rangle_{\mathrm{US}}$ differ only in their action at computation types. If the assumption that divergent behaviour is less than any convergent behaviour is made explicit, then an immediate connection can be made with one of the methods used to construct the lower and upper powerdomains [21,23].

We are now in a position to define the nine variants of similarity and bisimilarity. Six of the relations are defined as the greatest fixed points of combinations of the functions defined above. However, it is easy to verify by induction that the simple type system of the computational lambda-calculus ensures that the greatest fixed points of the functions are also least fixed points. The remaining relations, the mutual similarities, are the greatest symmetric relations contained in the three similarities.

Definition 3.2 The similarity and bisimilarity relations are defined by (where $\nu\mathcal{R}.\phi(R)$ denotes the greatest fixed point of ϕ):

$$\begin{aligned}
\lesssim_{\mathrm{LS}} &\stackrel{\mathrm{def}}{=} \nu\mathcal{R}.\langle\mathcal{R}\rangle_{\mathrm{LS}} \\
\lesssim_{\mathrm{US}} &\stackrel{\mathrm{def}}{=} \nu\mathcal{R}.\langle\mathcal{R}\rangle_{\mathrm{US}} \\
\lesssim_{\mathrm{CS}} &\stackrel{\mathrm{def}}{=} \nu\mathcal{R}.\langle\mathcal{R}\rangle_{\mathrm{LS}} \cap \langle\mathcal{R}\rangle_{\mathrm{US}} \\
\simeq_{\mathrm{LB}} &\stackrel{\mathrm{def}}{=} \nu\mathcal{R}.\langle\mathcal{R}\rangle_{\mathrm{LS}} \cap \langle\mathcal{R}^{\mathrm{op}}\rangle^{\mathrm{op}}_{\mathrm{LS}} \\
\simeq_{\mathrm{UB}} &\stackrel{\mathrm{def}}{=} \nu\mathcal{R}.\langle\mathcal{R}\rangle_{\mathrm{US}} \cap \langle\mathcal{R}^{\mathrm{op}}\rangle^{\mathrm{op}}_{\mathrm{US}} \\
\simeq_{\mathrm{CB}} &\stackrel{\mathrm{def}}{=} \nu\mathcal{R}.\langle\mathcal{R}\rangle_{\mathrm{LS}} \cap \langle\mathcal{R}\rangle_{\mathrm{US}} \cap \langle\mathcal{R}^{\mathrm{op}}\rangle^{\mathrm{op}}_{\mathrm{LS}} \cap \langle\mathcal{R}^{\mathrm{op}}\rangle^{\mathrm{op}}_{\mathrm{US}}
\end{aligned}$$

In addition, the mutual similarities $\simeq_{\mathrm{LS}}$, $\simeq_{\mathrm{US}}$, and $\simeq_{\mathrm{CS}}$ are defined to be the

greatest symmetric relations contained in $\lesssim_{\mathrm{LS}}$, $\lesssim_{\mathrm{US}}$, and $\lesssim_{\mathrm{CS}}$ respectively. The names of the relations are summarised in the table below.

	Lower	Upper	Convex
Similarity	$\lesssim_{\mathrm{LS}}$	$\lesssim_{\mathrm{US}}$	$\lesssim_{\mathrm{CS}}$
Mutual Similarity	$\simeq_{\mathrm{LS}}$	$\simeq_{\mathrm{US}}$	$\simeq_{\mathrm{CS}}$
Bisimilarity	$\simeq_{\mathrm{LB}}$	$\simeq_{\mathrm{UB}}$	$\simeq_{\mathrm{CB}}$

We refer to the tutorial papers [9,26] for the standard results concerning similarities and bisimilarities: each similarity is a preorder; each bisimilarity and mutual similarity is an equivalence; and the program that cannot converge, $\Omega \stackrel{\mathrm{def}}{=} \mathsf{fix}\, x.\, x$, is a least element for each of the similarities. In addition, it is immediate from the definitions that programs related by any of the similarities or bisimilarities have the same type.

Although the method of definition of the similarities and bisimilarities is convenient for the proof of compatibility in section 4, it is helpful to have the unwound definition to mind. In the case of convex bisimilarity we have that, if M and N are programs of the same computation type, then $M \simeq_{\mathrm{CB}} N$ if and only if:

(i) $\forall M_1.M \Downarrow^{\mathrm{may}} [M_1] \Longrightarrow (\exists N_1.N \Downarrow^{\mathrm{may}} [N_1] \wedge M_1 \simeq_{\mathrm{CB}} N_1)$; and

(ii) $\forall N_1.N \Downarrow^{\mathrm{may}} [N_1] \Longrightarrow (\exists M_1.M \Downarrow^{\mathrm{may}} [M_1] \wedge M_1 \simeq_{\mathrm{CB}} N_1)$; and

(iii) $M \Uparrow^{\mathrm{may}} \Longleftrightarrow N \Uparrow^{\mathrm{may}}$.

Lower bisimilarity follows the same pattern as convex bisimilarity with the exception that condition (iii) is dropped. We omit the unwinding of upper bisimilarity, but note that it identifies programs that can diverge, and that it does not identify a program that can diverge with one that does not. For example, the program in example 2.4 is identified with $?\mathbb{N}$ by lower similarity and bisimilarity, but not by the upper and convex variants of similarity and bisimilarity.

Lemmas 3.3 and 3.4 state elementary properties of, and relationships between, the different variants.

Lemma 3.3 *Erratic choice is the join operation for $\lesssim_{\mathrm{LS}}$, and the meet operation for $\lesssim_{\mathrm{US}}$ at the computation types, i.e., for all programs of a computation type L, M, N:*

(i) $(L \,\mathsf{or}\, M) \lesssim_{\mathrm{LS}} N$ *if and only if* $L \lesssim_{\mathrm{LS}} N$ *and* $M \lesssim_{\mathrm{LS}} N$.

(ii) $L \lesssim_{\mathrm{US}} (M \,\mathsf{or}\, N)$ *if and only if* $L \lesssim_{\mathrm{US}} M$ *and* $L \lesssim_{\mathrm{US}} N$.

Lemma 3.4 *The following inclusions hold:*

(i) $\lesssim_{\mathrm{CS}} \subseteq \lesssim_{\mathrm{LS}} \cap \lesssim_{\mathrm{US}}$, $\simeq_{\mathrm{CS}} \subseteq \simeq_{\mathrm{LS}} \cap \simeq_{\mathrm{US}}$, *and* $\simeq_{\mathrm{CB}} \subseteq \simeq_{\mathrm{LB}} \cap \simeq_{\mathrm{UB}}$.

(ii) $\simeq_{\mathrm{LB}} \subseteq \simeq_{\mathrm{LS}}$, $\simeq_{\mathrm{UB}} \subseteq \simeq_{\mathrm{US}}$, *and* $\simeq_{\mathrm{CB}} \subseteq \simeq_{\mathrm{CS}}$.

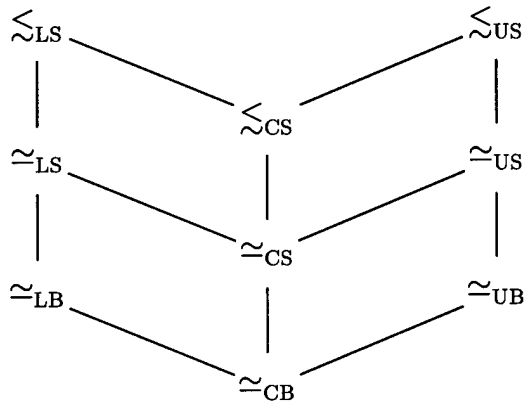


Fig. 4. Inclusions between Similarities and Bisimilarities

Example 3.5 *The following examples demonstrate the strictness of the inclusions of lemma 3.4:*

(i) *For any program* M, $(\Omega \text{ or } [[M]])$ *and* $(\Omega \text{ or } [(\Omega \text{ or } [M])])$ *are related by:* $(\simeq_{LB} \cap \simeq_{UB})$, $(\simeq_{LS} \cap \simeq_{US})$, *and* $(\lesssim_{LS} \cap \lesssim_{US})$; *but not by:* $\lesssim_{CS}$, $\simeq_{CS}$, *and* $\simeq_{CB}$. *From this we derive:*

$$(\Omega \text{ or } [[M]])\ ((\lesssim_{LS} \cap \lesssim_{US}) \setminus \lesssim_{CS})\ (\Omega \text{ or } [(\Omega \text{ or } [M])])$$
$$(\Omega \text{ or } [[M]])\ ((\simeq_{LS} \cap \simeq_{US}) \setminus \simeq_{CS})\ (\Omega \text{ or } [(\Omega \text{ or } [M])])$$
$$(\Omega \text{ or } [[M]])\ ((\simeq_{LB} \cap \simeq_{UB}) \setminus \simeq_{CB})\ (\Omega \text{ or } [(\Omega \text{ or } [M])])$$

(ii) *If* $M\ (\lesssim_{LS} \setminus \lesssim^{op}_{LS})\ N$, *then* $([M] \text{ or } [N])\ (\simeq_{LS} \setminus \simeq_{LB})\ [N]$. *Similarly, if we have* $M\ (\lesssim_{US} \setminus \lesssim^{op}_{US})\ N$, *then* $[M]\ (\simeq_{US} \setminus \simeq_{UB})\ ([M] \text{ or } [N])$. *The assignment* $M = \Omega$ *and* $N = [\star]$ *satisfies both of the hypotheses. Finally, if* $L\ (\lesssim_{CS} \setminus \lesssim^{op}_{CS})\ M\ (\lesssim_{CS} \setminus \lesssim^{op}_{CS})\ N$, *then:*

$$([L] \text{ or } ([M] \text{ or } [N]))\ (\simeq_{CS} \setminus \simeq_{CB})\ ([L] \text{ or } [N])$$

A suitable assignment is: $L = \Omega$, $M = (\Omega \text{ or } [\star])$, *and* $N = [\star]$.

Figure 4 depicts the relationships between the similarities and bisimilarities described in lemma 3.4 and example 3.5. Every edge denotes a strict inclusion.

As previously stated, the similarities and bisimilarities collapse to a similarity preorder and a bisimilarity equivalence respectively if we remove ?N from the programming language. It is easy to construct programs, see example 3.6, that demonstrate that the introduction of finite non-determinism is not conservative for any of the similarities and bisimilarities. Perhaps more surprising is that the upper and convex variants of similarity and bisimilarity are not conservatively extended when finite non-determinism is extended to countable non-determinism. This is discussed in example 3.7.

Example 3.6 *The following programs cannot be distinguished by application*

to deterministic programs:

$$\vdash \lambda x : \mathsf{P}(\mathsf{nat}).\, \mathsf{let}\, y \Leftarrow x\, \mathsf{in}\, [\mathsf{plus}\,(y, y)] : \mathsf{P}(\mathsf{nat}) \to \mathsf{P}(\mathsf{nat})$$
$$\vdash \lambda x : \mathsf{P}(\mathsf{nat}).\, \mathsf{let}\, y \Leftarrow x\, \mathsf{in}\, \mathsf{let}\, z \Leftarrow x\, \mathsf{in}\, [\mathsf{plus}\,(y, z)] : \mathsf{P}(\mathsf{nat}) \to \mathsf{P}(\mathsf{nat})$$

They can be distinguished by applying them to a non-deterministic program such as $(0\,\mathsf{or}\,1)$*, in which case the second program may converge to* $[\mathsf{plus}\,(0, 1)]$.

Example 3.7 *The following programs cannot be distinguished by application to finitely non-deterministic programs:*

$$\vdash \lambda^v x.\, [\star] : \mathsf{P}(\mathsf{nat}) \to \mathsf{P}(\mathsf{unit})$$
$$\vdash f\, 0 : \mathsf{P}(\mathsf{nat}) \to \mathsf{P}(\mathsf{unit})$$
$$\textit{where } f\, x\, y \stackrel{\mathrm{def}}{=} \mathsf{let}\, z \Leftarrow y\, \mathsf{in}\, \mathsf{if}\, (\mathsf{leq}\,(z, x))\, \mathsf{then}\, [\star]\, \mathsf{else}\, f\, z\, y$$

(the definition of f is intended to be formalised as in example 2.2). The programs can be distinguished by the upper and convex similarities and bisimilarities by applying them to $?\mathbb{N}$*. The first program is a strict constant function. The second program has only one convergent behaviour, will fail to terminate if its argument does, but, in addition, may diverge if it is possible to read an infinite, strictly increasing sequence of numbers from its argument.*

The similarity and bisimilarity relations extend in a standard way to relations on arbitrary typed terms by open extension. In general, the open extension of a relation on programs $\mathcal{R}$, denoted $\mathcal{R}^\circ$, relates typed terms $\Gamma \vdash M : \sigma$ and $\Gamma \vdash N : \sigma$ if $\Gamma = x_1 : \tau_1, \ldots, x_n : \tau_n$ and

$$M[L_1/x_1] \ldots [L_n/x_n] \;\; \mathcal{R} \;\; N[L_1/x_1] \ldots [L_n/x_n]$$

for all $L_1 \in Exp_{\tau_1}, \ldots, L_n \in Exp_{\tau_n}$.

4 Compatibility

In this section we sketch a proof that the open extensions of the similarities and bisimilarities of section 3 are compatible for the programming language. A relation $\mathcal{R}$ is compatible for a language if it is preserved by every constructor θ of the language, that is, $\mathcal{R}$ is closed under the rule:

$$\frac{M_1\, \mathcal{R}\, N_1 \;\ldots\; M_n\, \mathcal{R}\, N_n}{\theta(M_1, \ldots, M_n)\; \mathcal{R}\; \theta(N_1, \ldots, N_n)}$$

where the arity of θ is n. Compatibility is of fundamental importance because it is a prerequisite for compositional reasoning.

Howe [11] describes a method using a congruence candidate for proving the compatibility of lower similarity. In later work, Howe [12] and Ong [21] extend the method to convex bisimilarity and convex similarity respectively.

Unfortunately, other methods (see, e.g., [1,25,5]) that have been used to prove compatibility of similarity for deterministic programming languages do not seem to be applicable here: there are difficulties with interpreting ?$\mathbb{N}$ in the upper and convex powerdomains, and the methods based on syntactic logical relations use syntactic continuity (see section 5) to establish the fundamental property. Moreover, the compatibility of mutual similarity does not entail the compatibility of bisimilarity for a non-deterministic programming language.

We now sketch Howe and Ong's extension of Howe's congruence candidate method.

(i) The congruence candidate $\mathcal{R}^\bullet$ of a binary relation $\mathcal{R}$ on programs (which will range over the variants of similarity and bisimilarity) is an inductively defined binary relation on (potentially) open, well-typed terms. It is the least relation closed under the following rule, where θ ranges over constructors of the language, including variables, and the arity of θ is n:

$$\frac{L_1\,\mathcal{R}^\bullet\,M_1\;\ldots\;L_n\,\mathcal{R}^\bullet\,M_n \qquad \theta(M_1,\ldots,M_n)\,\mathcal{R}^\circ\,N}{\theta(L_1,\ldots,L_n)\,\mathcal{R}^\bullet\,N}$$

(ii) If $\mathcal{R}$ is a preorder, then the congruence candidate $\mathcal{R}^\circ$ satisfies:
   (a) $\mathcal{R}^\circ \subseteq \mathcal{R}^\bullet$.
   (b) $\mathcal{R}^\bullet;\mathcal{R}^\circ \subseteq \mathcal{R}^\bullet$.
   (c) $\mathcal{R}^\bullet$ is compatible.
   (d) $M_1\,\mathcal{R}^\bullet\,N_1 \wedge M_2\,\mathcal{R}^\bullet\,N_2 \Longrightarrow M_1[M_2/x]\,\mathcal{R}^\bullet\,N_1[N_2/x]$.

(iii) If $\mathcal{R}$ is a preorder, the restriction to programs $\mathcal{R}_0^\bullet$ of the congruence candidate $\mathcal{R}^\bullet$ is a post-fixed point of $\langle\cdot\rangle_{\mathrm{LS}}$ or $\langle\cdot\rangle_{\mathrm{US}}$ if $\mathcal{R}$ is:
   (a) $\mathcal{R} \subseteq \langle\mathcal{R}\rangle_{\mathrm{LS}} \Longrightarrow \mathcal{R}_0^\bullet \subseteq \langle\mathcal{R}_0^\bullet\rangle_{\mathrm{LS}}$.
   (b) $\mathcal{R} \subseteq \langle\mathcal{R}\rangle_{\mathrm{US}} \Longrightarrow \mathcal{R}_0^\bullet \subseteq \langle\mathcal{R}_0^\bullet\rangle_{\mathrm{US}}$.
   This is established by induction on the derivation of a may convergence judgement for (a), and on a natural number that is derived from a program that cannot diverge for (b)—although a problem is discussed below.

(iv) When $\mathcal{R}$ is lower, upper, or convex similarity, we deduce by co-induction that $\mathcal{R}_0^\bullet \subseteq \mathcal{R}$, and thus $\mathcal{R}^\bullet = \mathcal{R}^\circ$. Consequently, the open extensions of lower, upper, and convex similarity are compatible, because the respective congruence candidates are. Compatibility of the mutual similarities follows immediately.

(v) The final step is to deduce that each of the bisimilarities are compatible. If $\mathcal{R}$ is an equivalence, it can be shown using induction that $\mathcal{R}^\bullet \subseteq \mathcal{R}^{\bullet+\mathrm{op}}$, where $\mathcal{R}^{\bullet+}$ denotes the transitive closure of the congruence candidate of $\mathcal{R}$, which is compatible by an easy induction. Hence, $\mathcal{R}^{\bullet+} \subseteq \mathcal{R}^{\bullet+\mathrm{op}}$, so $\mathcal{R}^{\bullet+}$ is symmetric. In addition, we can derive from (iii) that:
   (a) $\mathcal{R} \subseteq \langle\mathcal{R}\rangle_{\mathrm{LS}} \Longrightarrow \mathcal{R}_0^{\bullet+} \subseteq \langle\mathcal{R}_0^\bullet\rangle_{\mathrm{LS}}^+ \subseteq \langle\mathcal{R}_0^{\bullet+}\rangle_{\mathrm{LS}}$.
   (b) $\mathcal{R} \subseteq \langle\mathcal{R}\rangle_{\mathrm{US}} \Longrightarrow \mathcal{R}_0^{\bullet+} \subseteq \langle\mathcal{R}_0^\bullet\rangle_{\mathrm{US}}^+ \subseteq \langle\mathcal{R}_0^{\bullet+}\rangle_{\mathrm{US}}$.
   As in (iv), co-induction can be used to show that $\mathcal{R}^{\bullet+}$ coincides with $\mathcal{R}$,

$$K \Downarrow^{\text{must}} \quad (K \in Can)$$

$$\frac{M \Downarrow^{\text{must}}}{\mathsf{uop}\,(M) \Downarrow^{\text{must}}} \qquad \frac{M \Downarrow^{\text{must}} \qquad N \Downarrow^{\text{must}}}{\mathsf{bop}\,(M, N) \Downarrow^{\text{must}}}$$

$$\frac{L \Downarrow^{\text{must}} \qquad M \Downarrow^{\text{must}}}{\mathsf{if}\ L\ \mathsf{then}\ M\ \mathsf{else}\ N \Downarrow^{\text{must}}} \quad (L \Downarrow^{\text{may}} n+1 \text{ and } n \in \mathbb{N})$$

$$\frac{L \Downarrow^{\text{must}} \qquad N \Downarrow^{\text{must}}}{\mathsf{if}\ L\ \mathsf{then}\ M\ \mathsf{else}\ N \Downarrow^{\text{must}}} \quad (L \Downarrow^{\text{may}} 0)$$

$$\frac{M \Downarrow^{\text{must}} \qquad M_1[N/x] \Downarrow^{\text{must}}}{M\,N \Downarrow^{\text{must}}} \quad (M \Downarrow^{\text{may}} \lambda x.\, M_1)$$

$$\frac{M \Downarrow^{\text{must}} \qquad \{N[M_1/x] \Downarrow^{\text{must}} \mid M \Downarrow^{\text{may}} [M_1]\}}{\mathsf{let}\ x \Leftarrow M\ \mathsf{in}\ N \Downarrow^{\text{must}}}$$

$$\frac{M[\mathsf{fix}\,x.\,M/x] \Downarrow^{\text{must}}}{\mathsf{fix}\,x.\,M \Downarrow^{\text{must}}} \qquad ?\mathbb{N} \Downarrow^{\text{must}}$$

Fig. 5. Must Convergence

when $\mathcal{R}$ is lower, upper, or convex bisimilarity. Therefore, the bisimilarities are compatible.

It is worth noting that the method also works for recursive types and in the absence of types, and that the use of the computational lambda-calculus means that we do not need to use disjoint sets of call-by-name and call-by-value variables as in [12,21].

However, we have glossed over a problem with (iii)(b). Howe and Ong assigned natural numbers to programs that cannot diverge *and* that have only finitely many convergent behaviours. For this reason, their proofs only hold for programming languages with finite non-determinism.

The method can be extended to a language with countable non-determinism by using induction on the derivation of a must convergence judgement. The rules for must convergence $\Downarrow^{\text{must}}$ appear in figure 5. Using induction on these rules, the proof works smoothly for both finite and countable non-determinism. The only problem is, how do we know that, for any program M, $M \Uparrow^{\text{may}}$ if and only if $M \Downarrow^{\text{must}}$? This turns out to be trivial, because the complement of the greatest fixed point of a monotone function on a complete boolean lattice is the least fixed point of another monotone function that can be derived from the original function (see [2]), and the must convergence rules in figure 5 are derived in this way from the may divergence rules in figure 3.

Theorem 4.1 *The lower, upper, and convex variants of similarity, mutual similarity, and bisimilarity are compatible.*

5 Convergence and Continuity

This section describes unwinding properties of recursive programs with respect to may and must convergence, and examines related syntactic continuity properties of the lower and upper similarities. The first part covers may convergence and lower similarity, and the second part covers must convergence and upper similarity. The latter includes an analysis of the heights, measured by ordinals, of derivation trees associated to must convergence judgements.

Well-typed terms of the form $\mathsf{fix}\, x.\, M$, henceforth called fixed point expressions, satisfy a finite unwinding property with respect to may convergence: for any fixed point expression $\mathsf{fix}\, x.\, M$, let $\mathsf{fix}^{(n)} x.\, M$ denote the n'th unwinding, defined inductively by:

$$\mathsf{fix}^{(0)} x.\, M \stackrel{\text{def}}{=} \Omega$$
$$\mathsf{fix}^{(n+1)} x.\, M \stackrel{\text{def}}{=} M[\mathsf{fix}^{(n)} x.\, M/x]$$

Then, whenever $x : \mathsf{P}(\sigma) \vdash M : \mathsf{P}(\sigma)$ and $x : \mathsf{P}(\sigma) \vdash N : \tau$,

$$N[\mathsf{fix}\, x.\, M/x] \Downarrow^{\text{may}} \text{ if and only if } \exists n < \omega.\ N[\mathsf{fix}^{(n)} x.\, M/x] \Downarrow^{\text{may}} \qquad (1)$$

where $L \Downarrow^{\text{may}}$ if and only if $\exists K.\, L \Downarrow^{\text{may}} K$. The proof is the same as for deterministic languages (see, e.g., [15,26]).

A related result is a so-called syntactic continuity property of lower similarity on deterministic programs: for terms N and M, as above, and $L \in Exp_\tau$,

$$N[\mathsf{fix}\, x.\, M/x] \lesssim_{\text{LS}} L \text{ if and only if } \forall n < \omega.\ N[\mathsf{fix}^{(n)} x.\, M/x] \lesssim_{\text{LS}} L \qquad (2)$$

See [14,26]. But syntactic continuity is not valid, in general, for non-deterministic programs:

Example 5.1 *Recall the program* $M \stackrel{\text{def}}{=} \mathsf{fix}\, x.\, ([0] \text{ or } \mathsf{let}\, y \Leftarrow x \,\mathsf{in}\, [\mathsf{plus}\,(y, 1)])$, *from example 2.4. Let* $N \stackrel{\text{def}}{=} \mathsf{let}\, x \Leftarrow ?\mathbb{N} \,\mathsf{in}\, [\mathsf{let}\, y \Leftarrow ?\mathbb{N} \,\mathsf{in}\, [\mathsf{if}\, \mathsf{leq}\,(x, y)\, \mathsf{then}\, x\, \mathsf{else}\, y]]$. *Then, for every finite unwinding* $M^{(n)}$ *of* M, $[M^{(n)}]$ *is lower similar to* N. *But* $[M]$ *and* N *are not lower similar. (The calculations are straightforward.)*

We now turn our attention to must convergence. First, consider finitely non-deterministic programs where non-determinism only occurs in the form of binary erratic choice. In this case, the derivation trees of the must convergence judgements introduced in section 4 are only finitely branching. As a result, the finite unwinding property of fixed point expressions (1) also holds with respect to must convergence. Moreover, upper similarity satisfies the syntactic continuity property (2) (see [15]).

In general, must convergence derivation trees of programs involving countable choice are countably branching. The complexity of the trees can be measured by assigning ordinals to them in the usual way (a node is assigned the supremum of the successors of the ordinals associated with its children,

see, e.g., [20]), and this allows us to give an ordinal bound to the induction used in the proof of theorem 4.1. The bound is simply the supremum of the ordinals associated to the derivations of must convergence judgements. Following work of Apt and Plotkin [3], the bound turns out to be ω_1^{CK}, the least non-recursive ordinal. We recall the definition of recursive ordinals below, but refer the reader to [29,20] for detailed accounts of the recursive ordinals.

Definition 5.2 An ordinal α is recursive if there exists a decidable order on the natural numbers that is order-isomorphic to α.

We first demonstrate that for each recursive ordinal α there is a program that cannot diverge, and that has a must convergence derivation tree with height at least α. Since α is a recursive ordinal, and it can be verified that every partial recursive function can be defined in the programming language, there is a program $M_\alpha : \mathsf{nat} \to \mathsf{nat} \to \mathsf{P(nat)}$ that does not diverge on any input, and the relation that it represents is order-isomorphic to α. Now we also need to construct a program slow that accepts as arguments a program representing an order on natural numbers, and a natural number. It then "counts down" from the given number until it reaches a minimal element, at which point it converges to $[\star]$. The type of the program is:

$$\vdash \mathsf{slow} : (\mathsf{nat} \to \mathsf{nat} \to \mathsf{P(nat)}) \to \mathsf{nat} \to \mathsf{P(unit)}$$

It is intended that the numeric argument, say n, is the code, with respect to the coding used by M_α, of an ordinal $\beta < \alpha$, and that the height of the derivation tree of $(\mathsf{slow}\, M_\alpha\, n) \Downarrow^{\mathrm{must}}$ is at least β. Intuitively, the must convergence derivation tree for this program should contain as sub-trees the derivation trees for $(\mathsf{slow}\, M_\alpha\, m) \Downarrow^{\mathrm{must}}$, where m codes an ordinal strictly less than β. The expressive power of $?\mathbb{N}$ can be used to do this: by choosing any natural number we are choosing the code of any ordinal strictly less than α. The decidability of the order on codes of ordinals strictly less than α allows us to then discard codes of ordinals that are greater than or equal to β. The following definition accomplishes this:

$$\mathsf{slow}\, f\, x \stackrel{\mathrm{def}}{=} \mathsf{let}\, y \Leftarrow ?\mathbb{N}\, \mathsf{in}\, \mathsf{let}\, z \Leftarrow f\, y\, x\, \mathsf{in}\, \mathsf{if}\, z\, \mathsf{then}\, (\mathsf{slow}\, f\, y)\, \mathsf{else}\, [\star]$$

Then, for each recursive ordinal α represented by M_α, we can define a program with a must convergence derivation tree of height α:

$$\mathsf{let}\, x \Leftarrow ?\mathbb{N}\, \mathsf{in}\, \mathsf{slow}\, M_\alpha\, x \tag{3}$$

In the other direction we have to show that the ordinal height of a must convergence derivation tree is always recursive. Suppose that M is a program that must converge, and that has a derivation tree with height α. The ordinals strictly less than α are represented by paths in the tree that start at the root of the tree, i.e., at M, together with annotations for the may convergence

side-conditions. With the side conditions given, it is decidable whether an arbitrary path is a valid path from M by checking each component of the path against the rule schema of figure 5. With a suitable encoding of paths in the tree as sequences of natural numbers, the derivation tree of $M \Downarrow^{\text{must}}$ is a recursive tree, and then the Kleene-Brouwer order on paths of the tree is both decidable and order-isomorphic to an ordinal greater than or equal to α. We refer the reader to [20] for the definition of recursive trees and the Kleene-Brouwer order.

In general, fixed point expressions in countably non-deterministic programs do not satisfy a finite unwinding property with respect to must convergence, because of the possibly transfinite heights of derivation trees; and the syntactic continuity property of upper similarity is invalid. For instance, if $\alpha \geq \omega$, the program (3) is a counterexample to both the finite unwinding and syntactic continuity properties. It is, however, possible to formulate an unwinding property for must convergence that holds for countable non-determinism by progressing to transfinite unwindings:

$$N[\mathsf{fix}\, x.\, M/x] \Downarrow^{\text{must}} \text{ if and only if } \exists \alpha < \omega_1^{\text{CK}}.\ N[\mathsf{fix}^{(\alpha)} x.\, M/x] \Downarrow^{\text{must}} \quad (4)$$

In order to make sense of this assertion, we need to define the transfinite unwindings and their must convergence behaviour. We extend the syntax with new terms $\mathsf{fix}^{(\lambda)} x.\, M$, for all recursive limit ordinals λ, with the same typing rule as for ordinary fixed point expressions. Arbitrary recursive unwindings $\mathsf{fix}^{(\alpha)} x.\, M$, for $\alpha < \omega_1^{\text{CK}}$, are defined if we let $\mathsf{fix}^{(0)} x.\, M \stackrel{\text{def}}{=} \Omega$, as above, and, inductively, $\mathsf{fix}^{(\alpha+1)} x.\, M \stackrel{\text{def}}{=} M[\mathsf{fix}^{(\alpha)} x.\, M/x]$, for all $\alpha < \omega_1^{\text{CK}}$. Next, the definition of must convergence has to be extended to the new terms. Intuitively, we want the following rule which expresses that the must convergence at recursive limit ordinals is the best of all the must convergence behaviours at smaller ordinals:

$$\frac{\mathsf{fix}^{(\alpha)} x.\, M \Downarrow^{\text{must}}}{\mathsf{fix}^{(\lambda)} x.\, M \Downarrow^{\text{must}}} \quad (\alpha < \lambda < \omega_1^{\text{CK}})$$

But, since the definition of must convergence in figure 5 depends on may convergence, it would be necessary to extend the may convergence relation on terms of computation type to the new terms as well, and it it is not clear how to do this. We get around this obstacle by giving a self-contained definition of must convergence at computation types without reference to may convergence. This can be achieved by means of either a "structurally inductive" definition of the must convergence predicate, $M \Downarrow^{\text{must}}$, in the style of Pitts [24] or an inductively defined must convergence relation, $M \Downarrow^{\text{must}} \mathcal{U}$, between terms M and sets of canonical terms $\mathcal{U}$. We sketch the second solution here. If M is a term in the original language, the meaning of $M \Downarrow^{\text{must}} \mathcal{U}$ is that M must converge and that $\mathcal{U}$ is the set of canonical terms that M may converge to.

For example,

$$K \Downarrow^{\text{must}} \{K\} \quad (K \in Can) \qquad\qquad ?\mathbb{N} \Downarrow^{\text{must}} \{[n] \,|\, n \in \mathbb{N}\}$$

A rule for let can be given without reference to may convergence:

$$\frac{M \Downarrow^{\text{must}} \mathcal{U} \qquad \{N[M_1/x] \Downarrow^{\text{must}} \mathcal{V}_{M_1} \,|\, [M_1] \in \mathcal{U}\}}{\mathsf{let}\, x \Leftarrow M \,\mathsf{in}\, N \Downarrow^{\text{must}} \bigcup \{\mathcal{V}_{M_1} \,|\, [M_1] \in \mathcal{U}\}}$$

The remaining rules are straightforward and make no reference to may convergence at computation types. The must convergence relation is extended to the new terms for transfinite unwindings of fixed point expressions by the rule:

$$\frac{\mathsf{fix}^{(\alpha)} x.\, M \Downarrow^{\text{must}} \mathcal{U}}{\mathsf{fix}^{(\lambda)} x.\, M \Downarrow^{\text{must}} \mathcal{U}} \quad (\alpha < \lambda < \omega_1^{\text{CK}})$$

The analysis of the definition of the must convergence predicate in figure 5 shows that the closure ordinal of the rules for the must convergence relation is also ω_1^{CK}. The must convergence predicate, $M \Downarrow^{\text{must}}$, is obtained from the must convergence relation as $M \Downarrow^{\text{must}} \stackrel{\text{def}}{\Leftrightarrow} \exists \mathcal{U}.\, M \Downarrow^{\text{must}} \mathcal{U}$. This concludes the definition of the must convergence behaviour of the transfinite unwindings of fixed point expressions. The proof of (4) is by induction on the derivation of the must convergence judgment.

The definition of the upper similarity from section 3 can be extended to programs with occurrences of transfinite unwindings of fixed point expressions, by extending $\langle \mathcal{R} \rangle_{\text{US}}$ to relate programs $M, N \in Exp_\sigma$ at computation types $\sigma = \mathsf{P}(\tau)$ if

$$\forall \mathcal{U}.M \Downarrow^{\text{must}} \mathcal{U} \Longrightarrow (\exists \mathcal{V}.N \Downarrow^{\text{must}} \mathcal{V} \wedge \forall N_1 \in \mathcal{V}.\exists M_1 \in \mathcal{U}.M_1 \,\mathcal{R}\, N_1)$$

The extension of upper similarity is obtained as the greatest fixed point of the extended definition of the function $\langle \cdot \rangle_{\text{US}}$. It is compatible with respect to the extended language. The compatibility proof for upper similarity from section 4 carries over if the induction is now conducted on the derivation of $M \Downarrow^{\text{must}} \mathcal{U}$.

We ask two questions about the extension of upper similarity to the extended language. First, is it a conservative extension, i.e., does it include the upper similarity relation defined in section 3 for the original language? Second, does it enjoy a transfinite syntactic continuity property? If both are answered affirmatively, we get a useful induction principle for reasoning about fixed point expressions with respect to upper similarity in the original language. The two questions are left as open problems.

6 Beyond Countable Choice

We have described two forms of non-determinism: the construct that we have taken as primitive ?ℕ, and binary erratic choice. In this section, we outline two other possibilities that have been proposed in the literature.

The first is based on the observation that binary erratic choice has precisely the same expressiveness as a new choice construct $?\{0,1\}$ that may converge to either [0] or [1], but cannot diverge. It is natural to ask whether other forms of non-determinism can be obtained in a similar way, e.g., if X is a non-empty set of natural numbers, then what is the expressiveness of a choice construct $?X$ that may converge to $[n]$, for any $n \in X$, but cannot diverge? It turns out that choice constructs for countably infinite sets of natural numbers are not always equally expressive, because Apt and Plotkin [3] show that exactly the choice constructs for non-empty, recursively enumerable sets of natural numbers can be defined from ?ℕ. This suggests that classifying non-deterministic programs by the cardinality of their convergent behaviour is misleading.

However, classification is not the only issue affected by the result of Apt and Plotkin. In the light of example 3.7, it is of interest to know whether the presence of additional forms of non-determinism further alters the upper and convex variants of similarity and bisimilarity. If this is the case, then a denotational model of non-determinism that can interpret sets of natural numbers that are not recursively enumerable will discriminate more than mutual similarity (or bisimilarity) for a programming language with only ?ℕ.

In order to study these problems, the programming language given here can be extended with additional choice constructs of the form described above. The proofs of compatibility sketched in section 4 readily extend to more general forms of "erratic" non-determinism [23]. Roscoe [30] studies similar non-deterministic choice constructs in an extension of CSP.

McCarthy's ambiguous choice operator exhibits a very different form of non-determinism. The ambiguous choice of two programs has a natural, fair (also known as dove-tailing) implementation: run both programs in parallel, and return the value of the first to converge. The ambiguous choice of two programs can converge to any value that the programs can converge to, but only diverges when both programs can diverge.

Moran [18] studies a functional programming language extended with ambiguous choice and proves that lower similarity is compatible for the language. An example is given there that shows that convex similarity cannot be compatible in the presence of ambiguous choice. Similar examples can be used to show that upper similarity and bisimilarity also fail to be compatible. However, the compatibility of convex bisimilarity in the presence of ambiguous choice is an open problem. The method described in section 4 is not immediately applicable because it would imply the compatibility of convex similarity.

7 Conclusion

We have defined a simply-typed functional programming language with an operator that can converge to any natural number, and have introduced nine compatible relations on programs. The relations are lower, upper, and convex variants of applicative similarity and bisimilarity. Although some of the relations have been studied individually in the literature, we have emphasised that they can be constructed using only two functions, and that this affords a natural structure to the proofs of compatibility. In addition, we have mapped the inclusions between the relations, and have given characteristic examples of the differences between them.

Although the programming language is based on the computational lambda-calculus and non-determinism is restricted to computation types, the examples can be modified for programming languages with non-determinism at function or product types (with the assumption that convergence is observable at those types). We also note that the mechanism for creating and composing computations in the computational lambda-calculus provides an alternative, with the same expressive power, to using both call-by-name and call-by-value abstractions to control the resolution of non-determinism.

A different, interesting example demonstrates that the upper and convex variants of similarity and bisimilarity are sensitive to whether finite or countable non-determinism is present in the programming language, i.e., countable non-determinism can be used to distinguish programs of function type that cannot be distinguished by finitely non-deterministic programs.

Previous proofs of compatibility have been restricted to languages with finite non-determinism. We have extended them to a programming language with countable non-determinism by using a relationship between least and greatest fixed points in complete boolean lattices to transform a co-inductively defined may divergence predicate into an inductively defined must convergence predicate. The supremum of the ordinal heights of the must convergence derivation trees is the least non-recursive ordinal ω_1^{CK}.

In this paper we have concentrated on operational models based on co-inductively defined similarity and bisimilarity relations. It may be argued that the resulting models are finer-grained than is warranted by reasonable notions of observation. An alternative is to operate with Morris-style contextual approximation preorders and equivalence relations which are naturally defined on the basis of the may and must convergence predicates [13,15]. The compatibility of the similarity and bisimilarity relations considered here implies that they are all contained in corresponding contextual relations. The inclusions are strict, for different reasons [15]. For instance, the failure of syntactic continuity in example 5.1 distinguishes lower similarity from may contextual approximation which does satisfy the syntactic continuity property (2) for arbitrary non-deterministic programs. Lower and upper similarity are used as auxiliary relations in [13] to reason about contextual equivalences

for the operationally-defined specification language of action semantics, action notation, which features countable non-determinism.

Acknowledgement

We would like to thank Ralph Loader, Peter Mosses, Luke Ong, Stan Wainer, and especially Andrew Moran for helpful conversations.

References

[1] S. Abramsky. The lazy lambda calculus. In D. A. Turner, editor, *Research Topics in Functional Programming*, The UT Year of Programming Series, pages 65–117. Addison-Wesley, 1990.

[2] P. Aczel. An introduction to inductive definitions. In J. Barwise, editor, *Handbook of Mathematical Logic*, number 90 in Studies in Logic. North-Holland Publishing Company, 1977.

[3] K. R. Apt and G. D. Plotkin. Countable nondeterminism and random assignment. *Journal of the ACM*, 33(4):724–767, October 1986.

[4] R. J. Bird and O. de Moor. *The Algebra of Programming.* Prentice Hall, 1997.

[5] L. Birkedal and R. Harper. Operational interpretations of recursive types in an operational setting (summary). In M. Abadi and T. Ito, editors, *Symposium on Theoretical Aspects of Computer Science, Sendai, Japan*, volume 1281 of *Lecture Notes in Computer Science.* Springer-Verlag, 1997.

[6] P. Cousot and R. Cousot. Inductive definitions, semantics and abstract interpretations. In *Conference Record of the 19th ACM Symposium on Principles of Programming Languages*, pages 83–94, 1992.

[7] A. D. Gordon. *Functional Programming and Input/Output.* Distinguished Dissertations in Computer Science. Cambridge University Press, 1994.

[8] A. D. Gordon. Bisimilarity as a theory of functional programming. BRICS Notes Series NS-95-3, Department of Computer Science, University of Aarhus, 1995.

[9] A. D. Gordon. A tutorial on co-induction and functional programming. In *Proceedings of the 1994 Glasgow Workshop on Functional Programming*, Workshops in Computing, 1995.

[10] C. A. Gunter. *Semantics of Programming Languages: Structures and Techniques.* Foundations of Computing. MIT Press, 1992.

[11] D. J. Howe. Equality in lazy computation systems. In *Proceedings, 4th Annual Symposium on Logic in Computer Science*, pages 198–203. Computer Society Press, Washington, 1989.

[12] D. J. Howe. Proving congruence of bisimulation in functional programming languages. *Information and Computation*, 124(2):103–112, 1996.

[13] S. B. Lassen. Action semantics reasoning about functional programs. *Math. Struct. in Comp. Science*, pages 557–589, 1997.

[14] S. B. Lassen. Relational reasoning about contexts. In A. D. Gordon and A. M. Pitts, editors, *Higher Order Operational Techniques in Semantics*, Publications of the Newton Institute, pages 91–135. Cambridge University Press, 1998.

[15] S. B. Lassen. *Relational Reasoning about Functions and Nondeterminism.* PhD thesis, Department of Computer Science, University of Aarhus, February 1998. URL `http://www.cl.cam.ac.uk/users/sbl21/docs/phd.html`.

[16] R. Milner. *Communication and Concurrency.* International Series in Computer Science. Prentice-Hall, New York, 1989.

[17] E. Moggi. Notions of computations and monads. *Information and Computation*, 93(1):55–92, 1991.

[18] A. Moran. Natural semantics for non-determinism. Licentiate thesis, Chalmers University of Technology and University of Göteborg, May 1994.

[19] P. D. Mosses. *Action Semantics.* Number 26 in Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1992.

[20] P. Odifreddi. *Classical Recursion Theory*, volume 125 of *Studies in Logic.* Elsevier Science Publishers B.V., 1989.

[21] C.-H. L. Ong. Concurrent lambda calculus, and a general pre-congruence theorem for applicative bisimulation. Preliminary version, August 1992.

[22] C.-H. L. Ong. Non-determinism in a functional setting. In *Proceedings, 8th Annual Symposium on Logic in Computer Science*, pages 275–286. Computer Society Press, Washington, 1993.

[23] C. S. Pitcher. *Functional Programming and Erratic Non-Determinism.* PhD thesis, Oxford University Computing Laboratory. In preparation (expected September 1998).

[24] A. M. Pitts. Parametric polymorphism and operational equivalence. Preliminary version. In this volume.

[25] A. M. Pitts. A note on logical relations between semantics and syntax. *Logic Journal of the Interest Group in Pure and Applied Logics*, 5(4):589–601, July 1997.

[26] A. M. Pitts. Operationally-based theories of program equivalence. In P. Dybjer and A. M. Pitts, editors, *Semantics and Logics of Computation.* Cambridge University Press, 1997. Lectures given at the CLICS-II Summer School on Semantics and Logics of Computation, Isaac Newton Institute for Mathematical Sciences, Cambridge, UK, September 1995.

[27] G. D. Plotkin. Domains. URL `http://hypatia.dcs.qmw.ac.uk/sites/other/domain.notes.other/`, 1983.

[28] S. Prasad, A. Giacalone, and P. Mishra. Operational and algebraic semantics for Facile: A symmetric integration of concurrent and functional programming. In M. S. Paterson, editor, *Automata, Languages and Programming*, volume 443 of *Lecture Notes in Computer Science*, pages 765–780. Springer-Verlag, 1990.

[29] H. Rogers, Jr. *Theory of Recursive Functions and Effective Computability.* McGraw-Hill Series in Higher Mathematics. McGraw-Hill, 1967.

[30] A. W. Roscoe. Two papers on CSP. Technical Report PRG-67, Programming Resarch Group, Oxford University Computing Laboratory, July 1988. (An alternative order for the failures model & Unbounded nondeterminism in CSP).

Electronic Notes in Theoretical Computer Science 10 (1998)
URL: http://www.elsevier.nl/locate/entcs/volume10.html 26 pages

Parametric Polymorphism and Operational Equivalence

(Preliminary Version)

Andrew M. Pitts [1]

Cambridge University Computer Laboratory
Pembroke Street, Cambridge CB2 3QG, UK

Abstract

Studies of the mathematical properties of impredicatively polymorphic types have for the most part focused on the *polymorphic lambda calculus* of Girard-Reynolds, which is a calculus of *total* polymorphic functions. This paper considers polymorphic types from a functional programming perspective, where the partialness arising from the presence of fixpoint recursion complicates the nature of potentially infinite ('lazy') datatypes. An operationally-based approach to Reynolds' notion of relational parametricity is developed for an extension of Plotkin's PCF with $\forall$-types and lazy lists. The resulting logical relation is shown to be a useful tool for proving properties of polymorphic types up to a notion of operational equivalence based on Morris-style contextual equivalence.

1 Introduction

> *'It turns out that virtually any basic type of interest can be encoded within* F_2 [polymorphic lambda calculus]. *Similarly, product types, sum types, existential types, and some recursive types, can be encoded within* F_2*: polymorphism has an amazing expressive power.'*
>
> Cardelli [6, page 2225]

It is a widely held view—typified by the above quotation—that the *polymorphic lambda calculus* (PLC) of Girard and Reynolds [10,32] plays a foundational role for the statics (type systems) of functional programming languages analogous to the one played by the untyped lambda calculus for the dynamics of such languages. The technical justification for this view rests on the encoding of a wide class of datatype constructions as PLC types: see for

[1] This research was funded by grant number GR/L38356 from the UK EPSRC.

example [5,34] and [11, Chapter 11]. However, these results cannot just be applied 'off the shelf' to deduce properties of functional programming languages equipped with polymorphic types. This is because PLC is a theory of *total* polymorphic functions—a consequence of the fact that β-reduction of typeable PLC terms is strongly normalising [10]. On the other hand, functional programming languages typically feature various mechanisms for making general forms of recursive definitions, both at the level of expressions and at the level of types. The first kind of definition of course entails the presence of 'partial' expressions, i.e. ones whose evaluation does not terminate. And then the second kind of definition may throw up types whose values involve partiality in complicated ways, through the use of non-strict constructors. Can such 'lazy' datatypes be encoded with combinations of function- and $\forall$-types?

A specific example may help to bring this question into sharper focus. Consider the type *num list* of lazy lists of numbers in a non-strict functional programming language, such as Haskell (www.haskell.org/report/). The canonical forms of this type are the nil-list, **nil**, and cons-expressions, $H{:}T$, where the head H (of type *num*) and the tail T (of type *num list* again) are not necessarily in canonical form (and therefore their evaluation may not terminate). Thus expressions of this type can represent finite lists (such as $0{:}\mathbf{nil}$), properly infinite lists (such as $\ell = 0{:}\ell$), or 'partial' lists (such as $0{:}\Omega$, where Ω is a divergent expression of type *num list*). Suppose now that the language is augmented with $\forall$-types. (We consider why one might want to do so in a moment.) In PLC, the type

$$L(\tau) \stackrel{\text{def}}{=} \forall\alpha\,(\alpha \to (\tau \to \alpha \to \alpha) \to \alpha) \quad (\alpha \text{ not free in } \tau)$$

encodes finite τ-lists—in the sense that the closed β-normal forms of $L(\tau)$ are in bijection with finite lists of closed β-normal forms of type τ. But what is the situation in the functional programming language? Can uses of the lazy list type *num list* always be replaced by the polymorphic type $L(num)$? More precisely, are *num list* and $L(num)$ 'operationally isomorphic', in the sense that there are functions in the language from *num list* to $L(num)$ and back again which are mutually inverse up to some reasonable notion of operational equivalence of expressions? Or is *num list* operationally isomorphic to some some other pure polymorphic type, or to no such type?

The reader will not find the answer to such questions in the literature, as far as I know. Partly this is because it is hard to construct denotational models of both impredicative polymorphism and fixpoint recursion. Such models do exist (see [7,8] for one style of model and [2] for another), but there is not much in the way of useful analysis of the properties of polymorphic types in them. On the other hand for pure PLC, Reynolds' notion of *relational parametricity* [33,20] turns out to provide a very powerful tool for such an analysis. There are models of PLC supporting a relationally parametric structure [3], and in such models polymorphic encodings of datatype constructions have strong properties, indeed have category-theoretic universal properties charac-

terising the constructions uniquely up to isomorphism [16,17,1,30]. Can one extend this relational approach to encompass fixpoint recursion? Unpublished work of Plotkin [29] indicates that one can. Here we show that a relatively simple, syntactic approach is possible.

Should one care? Well, for one thing the results presented here provide a basis for obtaining some 'free theorems' [35] up to operational equivalence (and modulo some restrictions to do with strictness) in languages like ML and Haskell that combine higher order functions, fixpoint recursion and *predicative* polymorphism. However, the power of the relational approach really shows when considering fully impredicative $\forall$-types. Since the type reconstruction problem is undecidable in this case [36] and explicit labelling with type information is considered cumbersome, most higher order typed languages meant for human programmers eschew fully impredicative polymorphism. However, it seems that impredicative polymorphism may be a useful feature of explicitly typed intermediate languages in compilers [14,23]. And undoubtedly there is foundational interest in knowing, in the presence of fixpoint recursion, to what extent various kinds of type can be reduced to pure polymorphic types.

As Wadler [35, Sec. 7] and Plotkin [29] point out, extending relational parametricity to cope with fixpoint recursion seems to necessitate working not with arbitrary relations, but with ones that are at least *admissible* in the domain-theoretic sense, i.e. that are bottom-relating and closed under taking limits of chains of related approximations. However, in this paper a relational framework for polymorphism and fixpoint recursion is developed which is based upon operational rather than denotational semantics. This allows one to avoid some of the complexities of the domain-theoretic approach. In particular, it turns out that questions of admissibility of relations only have to be treated implicitly, via an operationally-defined closure operator. This is perhaps the main technical contribution of the paper. As a result one obtains a straightforward and apparently quite powerful method for proving properties of Morris-style contextual equivalence of expressions and types involving impredicative polymorphism and fixpoint recursion; and one which is based only upon the syntax and operational semantics of the language. (See [25,26] for previous results of this kind.)

The plan of the paper is as follows. In the next section we introduce PCF^+, an extension of PCF [28] with lazy lists and $\forall$-types which will serve as the vehicle for examining the issues raised above. In Sec. 3 we define a notion of *observational congruence* for PCF^+ expressions: it is equivalent to a Morris-style contextual equivalence based upon observing convergence of evaluation in all contexts of list-type (but not of function- or $\forall$-type). Sec. 4 presents our syntactic version of relational parametricity. An action of the PCF^+ types on binary relations between PCF^+ expressions (of the same closed type) is defined. This gives rise to a certain binary logical relation which is shown to characterise PCF^+ observational congruence (Theorem 4.15). Sec. 5 shows how the logical relation can be used to prove basic properties of PCF^+

observational congruence (such as various extensionality properties) and to prove observational isomorphisms between types. We show, for example, that in PCF^+ it is indeed the case that $\alpha\, list$ is observationally isomorphic to the pure polymorphic type $\forall\,\alpha'\,(\alpha' \to (\alpha \to \alpha' \to \alpha') \to \alpha')$. Finally, Sec. 6 considers some directions in which the results presented here might usefully be extended.

2 Combining PCF and Impredicative Polymorphism

We will make use of a small programming language, PCF^+, which is a relative of that veteran of studies in programming languages, PCF [28]. Recall that PCF is a simply typed, call-by-name lambda calculus equipped with fixpoint recursion and some basic operations on ground types of natural numbers and booleans. To this we add $\forall$-types from the Girard-Reynolds polymorphic lambda calculus and a type constructor for lists. For reasons of parsimony we do without the ground types of natural numbers and booleans, because the role they play in the theory can be taken by the list types. So the PCF^+ types are given by

$$\begin{array}{lll} \tau ::= \alpha & & \text{type variable} \\ \quad \mid \tau\, list & & \text{list type} \\ \quad \mid \tau \to \tau & & \text{function type} \\ \quad \mid \forall\,\alpha\,(\tau) & & \forall\text{-type} \end{array}$$

and the PCF^+ terms are given by

$$\begin{array}{lll} M ::= x & & \text{variable} \\ \quad \mid \mathbf{nil}_\tau & & \text{empty list} \\ \quad \mid M :: M & & \text{non-empty list} \\ \quad \mid \mathbf{case}\, M\, \mathbf{of\ nil} \Rightarrow M \mid x :: x \Rightarrow M & & \text{case expression} \\ \quad \mid \lambda\, x : \tau\,(M) & & \text{function abstraction} \\ \quad \mid M\, M & & \text{function application} \\ \quad \mid \Lambda\,\alpha\,(M) & & \text{type generalisation} \\ \quad \mid M\,\tau & & \text{type specialisation} \\ \quad \mid \mathbf{fix}(M) & & \text{fixpoint recursion.} \end{array}$$

Here α and x range over disjoint countably infinite sets $TyVar$ and Var of *type variables* and *variables*, respectively. The constructions

$$\forall\,\alpha\,(-) \qquad \mathbf{case}\, M\, \mathbf{of\ nil} \Rightarrow M' \mid x :: x' \Rightarrow (-) \qquad \lambda\, x : \tau\,(-) \qquad \Lambda\,\alpha\,(-)$$

are binders and we will identify types and terms up to renaming of bound variables and bound type variables. We write $ftv(e)$ for the finite set of free

$$\Gamma, x:\tau \vdash x:\tau \qquad \Gamma \vdash \mathbf{nil}_\tau : \tau\ \mathit{list} \qquad \frac{\Gamma \vdash H:\tau \quad \Gamma \vdash T:\tau\ \mathit{list}}{\Gamma \vdash H :: T : \tau\ \mathit{list}}$$

$$\frac{\Gamma \vdash L:\tau_1\ \mathit{list} \quad \Gamma \vdash M_1:\tau_2 \quad \Gamma, h:\tau_1, t:\tau_1\ \mathit{list} \vdash M_2:\tau_2}{\Gamma \vdash \mathbf{case}\, L\, \mathbf{of\ nil} \Rightarrow M_1 \mid h :: t \Rightarrow M_2 : \tau_2}$$

$$\frac{\Gamma, x:\tau_1 \vdash M:\tau_2}{\Gamma \vdash \lambda x:\tau_1\,(M):\tau_1 \to \tau_2} \qquad \frac{\Gamma \vdash F:\tau_1 \to \tau_2 \quad \Gamma \vdash A:\tau_1}{\Gamma \vdash F\,A:\tau_2}$$

$$\frac{\alpha, \Gamma \vdash M:\tau}{\Gamma \vdash \Lambda\alpha\,(M):\forall\alpha\,(\tau)} \qquad \frac{\Gamma \vdash G:\forall\alpha\,(\tau_1)}{\Gamma \vdash G\,\tau_2 : \tau_1[\tau_2/\alpha]} \qquad \frac{\Gamma \vdash F:\tau \to \tau}{\Gamma \vdash \mathbf{fix}(F):\tau}$$

Fig. 1. PCF^+ type assignment relation

type variables of an expression e (be it a type or a term) and $fv(M)$ for the finite set of free variables of a term M. A type τ is *closed* if $ftv(\tau) = \emptyset$; whereas a term M is closed if $fv(M) = \emptyset$, whether or not it also has free type variables. The result of substituting a type τ for all free occurrences of a type variable α in e (a type or a term) will be denoted $e[\tau/\alpha]$. Similarly, $M[M'/x]$ denotes the result of substituting a term M' for all free occurrences of the variable x in M.

We are only interested in terms that can be assigned types. We use a typing judgement of the form $\Gamma \vdash M : \tau$ where

- the *typing environment* Γ is a pair A, Δ with A a finite subset of $TyVar$ and Δ a function defined on a finite subset $dom(\Delta)$ of Var and mapping each $x \in dom(\Delta)$ to a type with free type variables in A;
- M is a term with $ftv(M) \subseteq A$ and $fv(M) \subseteq dom(\Delta)$;
- τ is a type with $ftv(\tau) \subseteq A$.

Then the PCF^+ *type assignment relation* consists of all such judgements inductively defined by the axioms and rules in Fig. 1—all of which are quite standard. In the figure, the notation $\Gamma, x:\tau$ indicates the typing environment obtained from the typing environment $\Gamma = A, \Delta$ by properly extending the function Δ by mapping $x \notin dom(\Delta)$ to τ. Similarly, α, Γ is the typing environment obtained by properly extending A with an $\alpha \notin A$. Note that the explicit type information included in the syntax of terms means that, given Γ and M, there is at most one τ for which $\Gamma \vdash M : \tau$ holds.

We write Typ for the set of closed PCF^+ types. Given $\tau \in Typ$, we write $Term(\tau)$ for the set of PCF^+ terms M for which $\emptyset \vdash M : \tau$ is derivable from the axioms and rules in Fig. 1.

We give the operational semantics of PCF^+ in terms of an inductively defined relation of *evaluation*. It takes the form $M \Downarrow C$, where M and C are closed terms of the same closed type (i.e. $M, C \in Term(\tau)$ for some $\tau \in Typ$)

$$C \Downarrow C \quad (C \text{ canonical})$$

$$\frac{L \Downarrow \mathbf{nil}_\tau \quad M_1 \Downarrow C}{(\mathbf{case}\, L\, \mathbf{of\ nil} \Rightarrow M_1 \mid h :: t \Rightarrow M_2) \Downarrow C} \qquad \frac{L \Downarrow H :: T \quad M_2[H/h, T/t] \Downarrow C}{(\mathbf{case}\, L\, \mathbf{of\ nil} \Rightarrow M_1 \mid h :: t \Rightarrow M_2) \Downarrow C}$$

$$\frac{F \Downarrow \lambda x : \tau\,(M) \quad M[A/x] \Downarrow C}{F\,A \Downarrow C} \qquad \frac{G \Downarrow \Lambda\,\alpha\,(M) \quad M[\tau/\alpha] \Downarrow C}{G\,\tau \Downarrow C} \qquad \frac{F\,\mathbf{fix}(F) \Downarrow C}{\mathbf{fix}(F) \Downarrow C}$$

Fig. 2. PCF^+ evaluation relation

and where C is in *canonical form*:

$$C ::= \mathbf{nil}_\tau \mid M :: M \mid \lambda x : \tau\,(M) \mid \Lambda\,\alpha\,(M).$$

The evaluation relation is inductively defined by the axiom and rules in Fig. 2. Note that function application is given a call-by-name semantics and that the evaluation rule for type specialisations, $G\,\tau$, is dictated by our choice of canonical form at $\forall$-types—we choose not to evaluate 'under the Λ'. Evaluation is deterministic: given M, there is at most one C for which $M \Downarrow C$ holds; and of course the rule for **fix** entails that there may be no such C.

3 Observational Congruence

Recall that two terms of a programming language are regarded as *contextually equivalent* if they are interchangeable in any program without affecting the observable behaviour of the program upon execution. Of course, to make this a precise notion one has to choose what constitutes an executable program and what behaviour should be observable. For PCF, Plotkin [28] chooses 'program' to mean 'closed term of ground type' and the observable behaviour of such a program to be the constant (integer or boolean) to which it evaluates, if any. Since we have replaced ground types with list types, here we take a program to be a closed term of list type and we observe whether or not it evaluates to **nil**.

Thus given $\Gamma \vdash M_1 : \tau$ and $\Gamma \vdash M_2 : \tau$ in PCF^+, we can say that the terms M_1 and M_2 are contextually equivalent, and write $\Gamma \vdash M_1 =_{\mathrm{ctx}} M_2 : \tau$, if for any context $\mathcal{M}[-]$ for which $\mathcal{M}[M_1], \mathcal{M}[M_2] \in \mathit{Term}(\tau'\ \mathit{list})$ for some $\tau' \in \mathit{Typ}$, it is the case that

$$\mathcal{M}[M_1] \Downarrow \mathbf{nil}_{\tau'} \Leftrightarrow \mathcal{M}[M_2] \Downarrow \mathbf{nil}_{\tau'}.$$

As usual, a *context* $\mathcal{M}[-]$ means a PCF^+ term with a subterm replaced by the placeholder '$-$'; and then $\mathcal{M}[M']$ indicates the term that results from replacing the placeholder with the term M'. This is a textual substitution which may well involve capture of free variables in M' by binders in $\mathcal{M}[-]$. So, unlike terms, contexts are not identified up to renaming of bound variables. Although this might seem like a minor syntactic matter, it is an indication

$$\Gamma, x:\tau \vdash x \;\mathcal{E}\; x : \tau \qquad \Gamma \vdash \mathbf{nil}_\tau \;\mathcal{E}\; \mathbf{nil}_\tau : \tau\ \mathit{list} \qquad \frac{\Gamma \vdash H \;\mathcal{E}\; H' : \tau \quad \Gamma \vdash T \;\mathcal{E}\; T' : \tau\ \mathit{list}}{\Gamma \vdash (H :: T) \;\mathcal{E}\; (H' :: T') : \tau\ \mathit{list}}$$

$$\frac{\Gamma \vdash L \;\mathcal{E}\; L' : \tau_1\ \mathit{list} \quad \Gamma \vdash M_1 \;\mathcal{E}\; M_1' : \tau_2 \quad \Gamma, h:\tau_1, t:\tau_1\ \mathit{list} \vdash M_2 \;\mathcal{E}\; M_2' : \tau_2}{\Gamma \vdash (\mathbf{case}\, L\, \mathbf{of\ nil} \Rightarrow M_1 \mid h :: t \Rightarrow M_2) \;\mathcal{E}\; (\mathbf{case}\, L'\, \mathbf{of\ nil} \Rightarrow M_1' \mid h :: t \Rightarrow M_2') : \tau_2}$$

$$\frac{\Gamma, x:\tau_1 \vdash M \;\mathcal{E}\; M' : \tau_2}{\Gamma \vdash \lambda x:\tau_1.(M) \;\mathcal{E}\; \lambda x:\tau_1.(M') : \tau_1 \to \tau_2} \qquad \frac{\Gamma \vdash F \;\mathcal{E}\; F' : \tau_1 \to \tau_2 \quad \Gamma \vdash A \;\mathcal{E}\; A' : \tau_1}{\Gamma \vdash (F\,A) \;\mathcal{E}\; (F'\,A') : \tau_2}$$

$$\frac{\alpha, \Gamma \vdash M \;\mathcal{E}\; M' : \tau}{\Gamma \vdash \Lambda\alpha.(M) \;\mathcal{E}\; \Lambda\alpha.(M') : \forall\alpha.(\tau)} \qquad \frac{\Gamma \vdash G \;\mathcal{E}\; G' : \forall\alpha.(\tau_1)}{\Gamma \vdash (G\,\tau_2) \;\mathcal{E}\; (G'\,\tau_2) : \tau_1[\tau_2/\alpha]}$$

$$\frac{\Gamma \vdash F \;\mathcal{E}\; F' : \tau \to \tau}{\Gamma \vdash \mathbf{fix}(F) \;\mathcal{E}\; \mathbf{fix}(M') : \tau}$$

Fig. 3. Compatibility properties

$$\frac{\alpha, \Gamma \vdash M \;\mathcal{E}\; M' : \tau_1}{\Gamma[\tau_2/\alpha] \vdash M[\tau_2/\alpha] \;\mathcal{E}\; M'[\tau_2/\alpha] : \tau_1[\tau_2/\alpha]}$$

$$\frac{\Gamma, x:\tau_1 \vdash M \;\mathcal{E}\; M' : \tau_2}{\Gamma \vdash M[N/x] \;\mathcal{E}\; M'[N/x] : \tau_2} \quad \text{if } \Gamma \vdash N : \tau_1$$

Fig. 4. Substitutivity properties

that the notion of 'context' occurring in the above definition of contextual equivalence is rather too concrete. Perhaps a better indication is the fact that the *substitutivity property* of PCF^+ contextual equivalence

$$\Gamma, x:\tau_1 \vdash M =_{\mathrm{ctx}} M' : \tau_2 \;\&\; \Gamma \vdash N : \tau_1 \;\Rightarrow\; \Gamma \vdash M[N/x] =_{\mathrm{ctx}} M'[N/x] : \tau_2$$

is by no means an immediate consequence of the above definition of $=_{\mathrm{ctx}}$. This is because $M[N/x]$ is not of the form $\mathcal{M}_N[M]$ for some context $\mathcal{M}_N[-]$ (uniformly in M). Nevertheless, we can regard $M[N/x]$ as a use of M 'in context', or in other words, it is reasonable to demand that the above substitutivity property holds of a notion of PCF^+ contextual equivalence *by definition.*

For these reasons, in the rest of this section we develop a slightly more abstract treatment of PCF^+ contextual equivalence that avoids explicit use of contexts (following [12,18]). In fact, this approach also makes it easier to state and prove the fundamental properties of the logical relation to be defined in the next section.

Definition 3.1 Suppose $\mathcal{E}$ is a set of 4-tuples (Γ, M, M', τ) satisfying

(1) $$\Gamma \vdash M \mathcal{E} M' : \tau \Rightarrow (\Gamma \vdash M : \tau \ \& \ \Gamma \vdash M' : \tau)$$

where we write $\Gamma \vdash M \mathcal{E} M' : \tau$ instead of $(\Gamma, M, M', \tau) \in \mathcal{E}$.

(i) $\mathcal{E}$ is *compatible* if it is closed under the axioms and rules in Fig. 3. It is *substitutive* if it is closed under the rules in Fig. 4. (All these axioms and rules are intended to apply only to 4-tuples satisfying the well-formedness condition (1)).

(ii) Note that compatible relations are automatically reflexive. A PCF^+ *precongruence* is a compatible, substitutive relation which is also transitive. A PCF^+ *congruence* is a precongruence which is also symmetric.

(iii) $\mathcal{E}$ is *adequate* if for all closed types $\tau \in Typ$ and closed terms $M, M' \in Term(\tau\ list)$

$$\emptyset \vdash M \mathcal{E} M' : \tau\ list \Rightarrow (M \Downarrow \mathbf{nil}_\tau \Leftrightarrow M' \Downarrow \mathbf{nil}_\tau).$$

Theorem 3.2 (PCF^+ observational congruence) *There is a largest adequate PCF^+ congruence relation. We call it PCF^+* observational congruence *and write it as $=_{\mathrm{obs}}$.*

Proof. Obviously the intersection of any collection of PCF^+ congruence relations is another such. So the PCF^+ congruence relations form a complete lattice when ordered by subset inclusion. Slightly less obviously, but for quite general reasons, the join in this complete lattice of some congruences $\mathcal{E}_i$ is given by $(\bigcup_i \mathcal{E}_i)^*$, the reflexive-transitive closure of the set-theoretic union of the relations.[2] In particular, the join of all the adequate congruences is given by the reflexive-transitive closure of their union. But this is again adequate, because adequate relations clearly are closed under the operations of union and reflexive-transitive closure. □

PCF^+ observational congruence, $=_{\mathrm{obs}}$, is indeed equivalent to the contextual equivalence $=_{\mathrm{ctx}}$ which we mentioned at the start of this section.[3] However, this 'context free' version is technically more convenient when it comes to relating contextual equivalence to the logical relation introduced in the next section. For more results about 'context free' characterisations of contextual equivalence, see [19, Section 3.7].

We conclude this section with some examples of properties of PCF^+ types up to observational congruence. It does not seem easy to prove such properties directly from the definition of observational congruence (or using the more

[2] To see this, note that in the presence of reflexivity and transitivity, the compatibility conditions in Fig. 3 involving two hypotheses can each be replaced by two rules involving only single hypotheses: and of course the union of some relations closed under single-hypothesis rules (and the reflexive-transitive closure of such a relation) is another such.

[3] The only difficult part of the proof of coincidence of $=_{\mathrm{ctx}}$ and $=_{\mathrm{obs}}$ is the fact that $=_{\mathrm{ctx}}$ is closed under the rules in Fig. 4, the substitutivity properties. This can be proved as a corollary of the properties of the logical relation of Sec. 4, but we do not do so here.

concrete notion of contextual equivalence). The logical relation of the next section will provide the means to prove such properties.

In the case of closed terms of closed type, we just write $M_1 =_{\mathrm{obs}} M_2 : \tau$ for $\emptyset \vdash M_1 =_{\mathrm{obs}} M_2 : \tau$.

Example 3.3 (Polymorphic null type) Consider the type

$$\mathit{null} \stackrel{\mathrm{def}}{=} \forall\alpha.(\alpha).$$

In PCF^+ there is a closed term of this type, namely $\Omega \stackrel{\mathrm{def}}{=} \Lambda\alpha.(\mathbf{fix}(\lambda x : \alpha.(x)))$. This is a 'polymorphic bottom' since for each $\tau \in \mathit{Typ}$ it is not hard to see that $\Omega\tau$ diverges, i.e. that there is no C for which $\Omega\tau \Downarrow C$ holds. In fact, up to observational congruence, Ω is the only closed term of type *null*. In other words, we claim that for all $G \in \mathit{Term}(\mathit{null})$, one has $G =_{\mathrm{obs}} \Omega : \mathit{null}$.

Example 3.4 (Polymorphic unit type) Consider the type

$$\mathit{unit} \stackrel{\mathrm{def}}{=} \forall\alpha.(\alpha \to \alpha).$$

As well as the 'bottom' term $\Omega\,\mathit{unit}$, this type contains the polymorphic identity function $\Lambda\alpha.(\lambda x : \alpha.(x))$. But that is all: we claim that if $G \in \mathit{Term}(\mathit{unit})$, then either $G =_{\mathrm{obs}} (\Omega\,\mathit{unit}) : \mathit{unit}$ or $G =_{\mathrm{obs}} \Lambda\alpha.(\lambda x : \alpha.(x)) : \mathit{unit}$.

Example 3.5 (Polymorphic lists) Consider the polymorphic list type

$$L(\alpha) \stackrel{\mathrm{def}}{=} \forall\alpha'.(\alpha' \to (\alpha \to \alpha' \to \alpha') \to \alpha').$$

Define terms I and J as follows:

$$\begin{aligned} I \stackrel{\mathrm{def}}{=} \Lambda\alpha.(\mathbf{fix}(\lambda i : \alpha\,\mathit{list} \to L(\alpha).(\lambda \ell : \alpha\,\mathit{list}.(\\ \Lambda\alpha'.(\lambda x' : \alpha'.(\lambda f : \alpha \to \alpha' \to \alpha'.(\\ \mathbf{case}\,\ell\,\mathbf{of}\,\mathbf{nil} \Rightarrow x' \mid h :: t \Rightarrow f\,h(i\,t\,\alpha'\,x'\,f)))))))) \end{aligned}$$

$$J \stackrel{\mathrm{def}}{=} \Lambda\alpha.(\lambda p : L(\alpha).(p.(\alpha\,\mathit{list}).(N\,\alpha).(C\,\alpha)))$$

where $N \stackrel{\mathrm{def}}{=} \Lambda\alpha.(\mathbf{nil}_\alpha)$ and $C \stackrel{\mathrm{def}}{=} \Lambda\alpha.(\lambda h : \alpha.(\lambda t : \alpha\,\mathit{list}.(h :: t)))$. Then I and J are closed terms of types $\forall\alpha.(\alpha\,\mathit{list} \to L(\alpha))$ and $\forall\alpha.(L(\alpha) \to \alpha\,\mathit{list})$ respectively. We claim that these terms constitute an isomorphism between $\alpha\,\mathit{list}$ and $L(\alpha)$ up to observational congruence, polymorphically in α. In other words, the following observational congruences hold:

$$\alpha,\ \ell : \alpha\,\mathit{list} \vdash J\,\alpha.(I\,\alpha.\ell) =_{\mathrm{obs}} \ell : \alpha\,\mathit{list}$$
$$\alpha,\ g : L(\alpha) \vdash I\,\alpha.(J\,\alpha.g) =_{\mathrm{obs}} g : L(\alpha).$$

4 Syntactical Relational Parametricity

We aim to characterise PCF^+ observational congruence (defined in Theorem 3.2) in terms of a binary 'logical relation' incorporating a notion of relational parametricity analogous to that introduced by Reynolds [33] for the

(3) $\Delta_{\alpha_i}(\vec{r}) \stackrel{\text{def}}{=} r_i$

(4) $\Delta_{\tau\ list}(\vec{r}) \stackrel{\text{def}}{=} \nu r\,((1 + \Delta_\tau(\vec{r}) \times r)^{\top\top})$

(5) $\Delta_{\tau\to\tau'}(\vec{r}) \stackrel{\text{def}}{=} \Delta_\tau(\vec{r}) \to \Delta_{\tau'}(\vec{r})$

(6) $\Delta_{\forall\alpha(\tau)}(\vec{r}) \stackrel{\text{def}}{=} \forall r\,(\Delta_\tau(r^{\top\top}, \vec{r}))$

Note
(4) uses Definitions 4.9 and 4.7; (5) uses Definition 4.2; (6) uses Definitions 4.3 and 4.7.

Fig. 5. Definition of the logical relation Δ

pure polymorphic lambda calculus. The logical relation is parameterised by term-relations.

Definition 4.1 (Term-relations) Given closed PCF^+ types $\tau, \tau' \in Typ$, we write $Rel(\tau, \tau')$ for the set of subsets of $Term(\tau) \times Term(\tau')$.

Each open type $\tau(\alpha_1, \ldots, \alpha_n)$ gives rise to a function mapping tuples of term-relations to term-relations:

(2) $\quad r_1 \in Rel(\tau_1, \tau_1'), \ldots, r_n \in Rel(\tau_n, \tau_n') \mapsto \Delta_\tau(\vec{r}) \in Rel(\tau(\vec{\tau}), \tau(\vec{\tau'}))$.

This 'action' of PCF^+ types on term-relations is defined by induction on the structure of the type τ, as in Fig. 5. The definition makes use of various operations on term-relations, associated with the PCF^+ type constructors, which will be explained below. When τ is a closed type, (2) amounts to specifying a certain term-relation $\Delta_\tau \in Rel(\tau, \tau)$ and this will turn out to coincide with observational congruence:

$$M_1 =_{\text{ctx}} M_2 : \tau \;\Leftrightarrow\; (M_1, M_2) \in \Delta_\tau \qquad (M_1, M_2 \in Term(\tau)).$$

This, together with the definition of Δ at $\forall$-types is what permits us to deduce results like those in Examples 3.3–3.5 (see Secs 5.2–5.5).

The definition of $\Delta_{\tau_1\to\tau_2}$ in terms of Δ_{τ_1} and Δ_{τ_2} in Fig. 5 uses the following operation on term-relations, characteristic of the notion of 'logical relation' (cf. [27]).

Definition 4.2 (Action of $\to$ on term-relations) Given $r_1 \in Rel(\tau_1, \tau_1')$ and $r_2 \in Rel(\tau_2, \tau_2')$, we define $r_1 \to r_2 \in Rel(\tau_1 \to \tau_2, \tau_1' \to \tau_2')$ by:

$$(F, F') \in r_1 \to r_2 \stackrel{\text{def}}{\Leftrightarrow} \forall\,(A, A') \in r_1\,((F\,A, F'\,A') \in r_2).$$

Turning next to $\forall$-types, consider the following operation.

Definition 4.3 (Action of $\forall$ on term-relations) Let τ_1 and τ_1' be PCF^+ types with at most a single free type variable, α say. Suppose R is a function mapping term-relations $r \in Rel(\tau_2, \tau_2')$ (any $\tau_2, \tau_2' \in Typ$) to term-relations $R(r) \in Rel(\tau_1[\tau_2/\alpha], \tau_1'[\tau_2'/\alpha])$. Then we can form a term-relation $\forall r\,(R(r)) \in Rel(\forall\,\alpha\,(\tau_1), \forall\,\alpha\,(\tau_1'))$ as follows:

$$\Gamma \vdash Id : \tau \multimap \tau$$

$$\frac{\Gamma \vdash S : \tau' \multimap \tau''}{\Gamma \vdash S \circ (\mathbf{case} - \mathbf{of\ nil} \Rightarrow M_1 \mid h :: t \Rightarrow M_2) : \tau\ \mathit{list} \multimap \tau''} \quad \text{if} \begin{cases} \Gamma \vdash M_1 : \tau' \\ \Gamma, h : \tau, t : \tau\ \mathit{list} \vdash M_2 : \tau' \end{cases}$$

$$\frac{\Gamma \vdash S : \tau' \multimap \tau''}{\Gamma \vdash S \circ (-A) : (\tau \to \tau') \multimap \tau''} \quad \text{if } \Gamma \vdash A : \tau \qquad \frac{\Gamma \vdash S : \tau'[\tau/\alpha] \multimap \tau''}{\Gamma \vdash S \circ (-\tau) : \forall \alpha\,(\tau') \multimap \tau''}$$

Fig. 6. Typing frame stacks

$$(G, G') \in \forall r\,(R(r)) \overset{\text{def}}{\Leftrightarrow} \forall \tau_2, \tau_2' \in \mathit{Typ}\,(\forall r \in \mathit{Rel}(\tau_2, \tau_2')\,((G\,\tau_2, G'\,\tau_2') \in R(r))).$$

From [33] one might expect the definition of $\Delta_{\forall \alpha\,(\tau_1)}(\vec{r})$ to be $\forall r\,(\Delta_{\tau_1}(r, \vec{r}))$. This will not do for PCF^+ because of the presence of fixpoint recursion. For then we would have $\Delta_{\forall \alpha\,(\alpha)} = \forall r\,(r) = \emptyset$ (since we can instantiate the parameter r with the empty relation). But then $\Delta_{\forall \alpha\,(\alpha)}$ cannot coincide with $=_{\mathrm{obs}}$ as we desire, because the latter is not empty: from Example 3.3 we have $\Omega =_{\mathrm{obs}} \Omega : \forall \alpha\,(\alpha)$. As this example may indicate, we will have to restrict the parameterising relations in the definition of $\Delta_{\forall \alpha\,(\tau)}$ at least to be 'admissible for fixpoint induction', in some way. In domain theory, a subset of a domain is said to be *admissible* if it contains the least element of the domain and is closed under taking least upper bounds of chains in the domain. It is perfectly possible to make use of a direct, syntactic version of this notion by considering term-relations that are closed under certain syntactically definable chains and their limits, e.g. those generated by the finite unfoldings of a fixpoint term, or by syntactically definable projection functions. See [4] for an example of this approach to 'syntactic admissibility'. Here we take a more indirect approach, already present implicitly in [26]. It enables us to obtain the necessary admissibility properties as a corollary of a construction that we need anyway in order to build sufficient properties of evaluation into the logical relation for it to characterise observational congruence. The key idea is to consider relations between PCF^+ *evaluation contexts* [9]—those contexts $\mathcal{M}[-]$ with a single occurrence of the placeholder, '$-$', in the position where the next subexpression will be evaluated. To aid analysis of the termination relation $M \Downarrow \mathbf{nil}_\tau$, we use the following reformulation of evaluation contexts as stacks of 'evaluation frames' (cf. [15] and [26, Sec. 3]).

Definition 4.4 (Frame Stacks) The grammar for PCF^+ *frame stacks* is

$$S ::= Id \mid S \circ F$$

where F ranges over *frames*:

$$F ::= (\mathbf{case} - \mathbf{of\ nil} \Rightarrow M \mid x :: x \Rightarrow M) \mid (-M) \mid (-\tau).$$

We use the judgement $\Gamma \vdash S : \tau \multimap \tau'$ to indicate the argument and result

type of a frame stack. Here Γ is a typing environment, as defined in Sec. 2, and we assume similar well-formedness conditions as there (free variables and free type variables of all expressions in the judgement are listed in Γ). The axiom and rules inductively defining this judgement are given in Fig. 6. Unlike PCF^+ terms, we have not included explicit type information in the syntax of frame stacks. For example, *Id* is not tagged with a type. However, it is not hard to see that, given Γ, S, and τ, there is at most one τ' for which $\Gamma \vdash S : \tau \multimap \tau'$ holds. This property is enough for our purposes, since the argument type of a frame stack will always be supplied in any particular situation in which we use it.

Definition 4.5 Given closed PCF^+ types $\tau, \tau' \in Typ$, we write $Stack(\tau, \tau')$ for the set of frame stacks S for which $\emptyset \vdash S : \tau \multimap \tau'$ is derivable from the axiom and rules in Fig. 6.

The analogue for frame stacks of the operation of filling the hole of an evaluation context with a term is given by the operation $S, M \mapsto S@M$, of applying a frame stack to term. It is defined by induction on the length of the stack:

$$\begin{aligned} Id@M &\stackrel{\text{def}}{=} M \\ (S \circ F)@M &\stackrel{\text{def}}{=} S@(F[M/-]) \end{aligned}$$

where $F[M/-]$ is the term that results from replacing '$-$' by M in the frame F. Note that if $S \in Stack(\tau, \tau')$ and $M \in Term(\tau)$, then $S@M \in Term(\tau')$.

Theorem 4.6 (A structural induction principle for termination) *Given a closed PCF^+ term M of list type, $M \in Term(\tau\ list)$ say, write $M \Downarrow$ to mean that $M \Downarrow \mathbf{nil}_\tau$ is derivable from the axiom and rules for evaluation given in Fig. 2. Then for all $\tau, \tau' \in Typ$, $M \in Term(\tau)$ and $S \in Stack(\tau, \tau'\ list)$ we have*

$$S@M \Downarrow \;\Leftrightarrow\; S \top M$$

where the relation $(-) \top (-)$ is inductively defined by the axiom and rules in Fig. 7. If $S \top M$ holds we say that S and M are coterminate.

The proof of this theorem is quite straightforward and is omitted. Not only does the $- \top -$ relation facilitate inductive proofs involving termination, but also it is the key to our syntactic treatment of admissibility, as we now explain. Given a closed PCF^+ type $\tau \in Typ$, define

$$Term^\top(\tau) \stackrel{\text{def}}{=} \bigcup_{\tau' \in Typ} Stack(\tau, \tau'\ list).$$

We write $Rel^\top(\tau, \tau')$ for the set of subsets of $Term^\top(\tau) \times Term^\top(\tau)$ and refer to such subsets as *stack-relations*. Using the $(-) \top (-)$ relation from Theorem 4.6 we can manufacture a stack-relation from a term-relation and vice versa, as follows.

$$Id \top \mathbf{nil}_\tau \qquad \frac{S \top M_1}{S \circ (\mathbf{case}\ -\ \mathbf{of\ nil} \Rightarrow M_1 \mid h :: t \Rightarrow M_2) \top \mathbf{nil}_\tau}$$

$$\frac{S \top M_2[H/h, T/t]}{S \circ (\mathbf{case}\ -\ \mathbf{of\ nil} \Rightarrow M_1 \mid h :: t \Rightarrow M_2) \top H :: T}$$

$$\frac{S \top M[A/x]}{S \circ (-A) \top \lambda x : \tau\,(M)} \qquad \frac{S \top M[\tau/\alpha]}{S \circ (-\tau) \top \Lambda \alpha\,(M)}$$

$$\frac{S \circ (\mathbf{case}\ -\ \mathbf{of\ nil} \Rightarrow M_1 \mid h :: t \Rightarrow M_2) \top L}{S \top \mathbf{case}\, L\, \mathbf{of\ nil} \Rightarrow M_1 \mid h :: t \Rightarrow M_2} \qquad \frac{S \circ (-A) \top F}{S \top F\, A}$$

$$\frac{S \circ (-\tau) \top G}{S \top G\tau} \qquad \frac{S \circ (-\mathbf{fix}(F)) \top F}{S \top \mathbf{fix}(F)}$$

Fig. 7. PCF^+ **nil**-termination relation

Definition 4.7 (The $(-)^\top$ operation on relations) Given any $\tau, \tau' \in Typ$ and $r \in Rel(\tau, \tau')$ define $r^\top \in Rel^\top(\tau, \tau')$ by

$$(S, S') \in r^\top \overset{\text{def}}{\Leftrightarrow} \forall\, (M, M') \in r\, (S \top M \Leftrightarrow S' \top M')$$

and given any $s \in Rel^\top(\tau, \tau')$ define $s^\top \in Rel(\tau, \tau')$ by

$$(M, M') \in s^\top \overset{\text{def}}{\Leftrightarrow} \forall\, (S, S') \in s\, (S \top M \Leftrightarrow S' \top M').$$

Just from the form of the definition of the operations $r \mapsto r^\top$, $s \mapsto s^\top$ (i.e. without using any properties of the termination relation $(-) \top (-)$) it is clear that one has a Galois connection:

(7) $r_1 \subseteq r_2 \Rightarrow (r_2)^\top \subseteq (r_1)^\top$
(8) $s_1 \subseteq s_2 \Rightarrow (s_2)^\top \subseteq (s_1)^\top$
(9) $r \subseteq s^\top \Leftrightarrow s \subseteq r^\top$.

So in particular $r \mapsto r^{\top\top}$ is a closure operator for term-relations, i.e. is order-preserving, inflationary and idempotent. Thus we say that a term-relation r is $\top\top$-*closed* if $r = r^{\top\top}$, or equivalently if $r^{\top\top} \subseteq r$, or equivalently if $r = s^\top$ for some stack-relation s, or equivalently if $r = (r')^{\top\top}$ for some term-relation r'. Note that the use of $(-)^{\top\top}$ in clause (6) of Fig. 5 means that the universal quantification over term-relations in the definition of $\Delta_{\forall \alpha\,(\tau)}(\vec{r})$ is being restricted to range over $\top\top$-closed relations.

The next result is an indication that $\top\top$-closed term-relations have appropriate 'admissibility' properties.

Theorem 4.8 (Admissibility of $\top\top$-closed term-relations) *Suppose $r \in Rel(\tau, \tau')$ is $\top\top$-closed. Then for any $F \in Term(\tau \to \tau)$ and*

$F' \in Term(\tau' \to \tau')$ one has

$$(F, F') \in r \to r \;\Rightarrow\; (\mathbf{fix}(F), \mathbf{fix}(F')) \in r.$$

Proof. We use the following

Unwinding Theorem. *For any $\tau \in Typ$, $F \in Term(\tau \to \tau)$, $S \in Term^{\top}(\tau)$, defining $\mathbf{fix}^{(0)}(F) \stackrel{\text{def}}{=} \Omega\tau$ and $\mathbf{fix}^{(n+1)}(F) \stackrel{\text{def}}{=} F\,\mathbf{fix}^{(n)}(F)$, it is the case that*

$$S \top \mathbf{fix}(F) \;\Leftrightarrow\; \exists n \in \mathbb{N}(S \top \mathbf{fix}^{(n)}(F)).$$

This result, or rather a slight generalisation of it using fixpoint terms in arbitrary contexts, can be proved by relatively straightforward inductions over the definition of the $(-) \top (-)$ relation. We omit the details (but see for example [26, Theorem 3.2]).

It is not hard to see that $S \top \Omega\tau$ does not hold for any $S \in Term^{\top}(\tau)$ (since evaluation of $\Omega\tau$ never terminates). Thus $(\Omega\tau, \Omega\tau') \in s^{\top}$, for any $s \in Rel^{\top}(\tau, \tau')$. Hence in particular taking $s = r^{\top}$, we have

$$(\mathbf{fix}^{(0)}(F), \mathbf{fix}^{(0)}(F')) \in r^{\top\top} = r.$$

So if $(F, F') \in r \to r$, it follows by induction on n that

$$(\mathbf{fix}^{(n)}(F), \mathbf{fix}^{(n)}(F')) \in r$$

holds for all $n \in \mathbb{N}$. Finally, for any $(S, S') \in r^{\top}$ we have

$$\begin{aligned}
S \top \mathbf{fix}(F) &\Leftrightarrow \exists n \in \mathbb{N}(S \top \mathbf{fix}^{(n)}(F)) && \text{by the Unwinding Theorem} \\
&\Leftrightarrow \exists n \in \mathbb{N}(S' \top \mathbf{fix}^{(n)}(F')) && \text{since } (\mathbf{fix}^{(n)}(F), \mathbf{fix}^{(n)}(F')) \in r \\
& && \text{and } (S, S') \in r^{\top} \\
&\Leftrightarrow S' \top \mathbf{fix}(F') && \text{by the Unwinding Theorem.}
\end{aligned}$$

Thus by definition of $(-)^{\top}$, $(\mathbf{fix}(F), \mathbf{fix}(F)) \in r^{\top\top} = r$, as required. □

To complete the explanation of Fig. 5 we have to define the action of the *list* type constructor on term-relations.

Definition 4.9 (Action of $(-)$ *list* on term-relations)
Given $\tau, \tau' \in Typ$, $r_1 \in Rel(\tau, \tau')$ and $r_2 \in Rel(\tau\ list, \tau'\ list)$, define $1 + (r_1 \times r_2) \in Rel(\tau\ list, \tau'\ list)$ by:

$$\begin{aligned}
&1 + (r_1 \times r_2) \stackrel{\text{def}}{=} \\
&\quad \{(\mathbf{nil}_{\tau}, \mathbf{nil}_{\tau'})\} \cup \{(H :: T, H' :: T') \mid (H, H') \in r_1 \;\&\; (T, T') \in r_2\}.
\end{aligned}$$

Note that the subset relation makes $Rel(\tau\ list, \tau'\ list)$ into a complete lattice and that, for each r_1, the function $r_2 \mapsto (1 + (r_1 \times r_2))^{\top\top}$ is monotone. Therefore we can form its greatest fixed point, $\nu r\,((1 + (r_1 \times r))^{\top\top})$. The function

$$r_1 \in Rel(\tau, \tau') \;\mapsto\; \nu r\,((1 + (r_1 \times r))^{\top\top}) \in Rel(\tau\ list, \tau'\ list)$$

is the action of $(-)$ *list* on term-relations used in clause (4) of Fig. 5 to define $\Delta_{\tau\ list}$ in terms of Δ_{τ}.

Remark 4.10 (List bisimulations) When r_1 is $\top\top$-closed, one can give an alternative characterisation of $\nu\, r\,((1 + (r_1 \times r))^{\top\top})$ which accords more closely with the characterisation of observational congruence of lazy lists in terms of a notion of *bisimilarity* to be found, for example, in [24, Sec. 3]. Given $r_1 \in Rel(\tau, \tau')$, call a term-relation $r_2 \in Rel(\tau\ list, \tau'\ list)$ an r_1-*simulation* if it satisfies that whenever $(L, L') \in r_2$ then

- if $L \Downarrow \mathbf{nil}_\tau$, then $L' \Downarrow \mathbf{nil}_{\tau'}$
- if $L \Downarrow H :: T$, then for some H' and T' it is the case that $L' \Downarrow H' :: T'$ with $(H, H) \in r_1$ and $(T, T') \in r_2$.

Say that r_2 is an r_1-*bisimulation* if both it and its reciprocal relation $(r_2)^{op} \stackrel{\text{def}}{=} \{(L, L') \mid (L', L) \in r_2\}$ are r_1-simulations. Then one can prove that *when* r_1 *is* $\top\top$*-closed,* $\nu\, r\,((1 + (r_1 \times r))^{\top\top})$ *is the greatest* r_1*-bisimulation.* (Moreover we will see next that the r_1 used in Fig. 5, namely $\Delta_\tau(\vec{r})$, is always $\top\top$-closed provided the term-relations $\vec{r}$ are.)

The following properties of $\top\top$-closed term-relations will be needed below.

Lemma 4.11*(i) If* r_2 *is* $\top\top$*-closed, then so is* $r_1 \to r_2$*, for any* r_1*. If* R *is as in Definition 4.3 and each* $R(r)$ *is* $\top\top$*-closed, then so is* $\forall\, r\,(R(r))$*. Hence it follows by induction on the structure of PCF*$^+$ *types* τ *that* $\Delta_\tau(\vec{r})$ *is* $\top\top$*-closed provided the term-relations* $\vec{r}$ *are. (The induction step for list types is automatic, because* $\Delta_{\tau\ list}(\vec{r})$ *is a term-relation* r *satisfying* $r = (1 + \Delta_\tau(\vec{r}) \times r)^{\top\top}$ *(it is the greatest such) and hence it is always* $\top\top$*-closed.)*

(ii) Kleene equivalence, $=_{\text{kl}}$, *is defined by:*

$$M_1 =_{\text{kl}} M_2 : \tau \stackrel{\text{def}}{\Leftrightarrow} \forall\, C\,(M_1 \Downarrow C \;\Leftrightarrow\; M_2 \Downarrow C).$$

Then if $r \in Rel(\tau, \tau')$ *is a* $\top\top$*-closed term-relation, one has*

$$M_1 =_{\text{kl}} M_2 : \tau \;\;\&\;\; (M_1, M_1') \in r \;\;\&\;\; M_1' =_{\text{kl}} M_2' : \tau' \;\Rightarrow\; (M_2, M_2') \in r.$$

Fig. 5 defines a family of binary relations between *closed* terms. We extend this to a relation between open terms, of the form considered in Definition 3.1, by considering closing substitutions.

Definition 4.12 (Logical relation on open terms) Suppose $\Gamma \vdash M : \tau$ and $\Gamma \vdash M' : \tau$ hold, with $\Gamma = \alpha_1, \ldots, \alpha_m,\ x : \tau_1, \ldots, x : \tau_n$ say. Write

$$\Gamma \vdash M\ \Delta\ M' : \tau \tag{10}$$

to mean:

given any $\sigma_i, \sigma_i' \in Typ$ and $r_i \in Rel(\sigma_i, \sigma_i')$ (for $i = 1, \ldots, m$) with each r_i $\top\top$-closed, then for any $(N_j, N_j') \in \Delta_{\tau_j}(\vec{r})$ (for $j = 1, \ldots, n$) it is the case that $(M[\vec{\sigma}/\vec{\alpha}, \vec{N}/\vec{x}], M'[\vec{\sigma'}/\vec{\alpha}, \vec{N'}/\vec{x}]) \in \Delta_\tau(\vec{r})$

(The restriction to $\top\top$-closed relations in this definition accords with the definition of Δ at $\forall$-types.)

Theorem 4.13 (Fundamental properties of the logical relation) *With Δ extended to open terms as in the previous definition, we have:*

(i) Δ is compatible (cf. Definition 3.1(i)).

(ii) For each closed type $\tau \in Typ$ we have

$$(M, M) \in \Delta_\tau \qquad \text{and} \qquad (S, S) \in (\Delta_\tau)^\top$$

for all closed terms $M \in Term(\tau)$ and frame stacks $S \in Term^\top(\tau)$.

(iii) Δ is substitutive (cf. Definition 3.1(i)).

Proof. For part (i), one has to prove that (10) is closed under the axioms and rules in Fig. 3. Most of these compatibility properties are immediate consequences of the definition of Δ in Fig. 5 and the way it is extended to open terms in Definition 4.12. However, those for **case**, $\lambda x : \tau\,(-)$, and $\Lambda\,\alpha\,(-)$ require Lemma 4.11 together with the following Kleene equivalences:

$$\begin{aligned}
(\lambda x : \tau_1\,(M))\,A &=_{\mathrm{kl}} M[A/x] : \tau_2 \\
(\Lambda\,\alpha\,(M))\,\tau_2 &=_{\mathrm{kl}} M[\tau_2/\alpha] : \tau_1[\tau_2/\alpha] \\
\mathbf{case}\ \mathbf{nil}_{\tau_1}\ \mathbf{of}\ \mathbf{nil} \Rightarrow M_1 \mid h :: t \Rightarrow M_2 &=_{\mathrm{kl}} M_1 : \tau_2 \\
\mathbf{case}\ H :: T\ \mathbf{of}\ \mathbf{nil} \Rightarrow M_1 \mid h :: t \Rightarrow M_2 &=_{\mathrm{kl}} M_2[H/h, T/t] : \tau_2.
\end{aligned}$$

The compatibility condition for **fix** follows from Theorem 4.8 (together with Lemma 4.11(i)).

Part (ii) follows from part (i). Since the relation (10) is compatible, it is automatically reflexive and hence in particular $(M, M) \in \Delta_\tau$ holds. The fact that one also has $(S, S) \in (\Delta_\tau)^\top$ can be proved by induction on the structure of S, using compatibility properties that form part of the proof of part (i).

For part (iii), one has to prove that the relation (10) is closed under the axioms and rules in Fig. 4. The type-substitutivity property reduces to showing for open types $\tau(\vec{\alpha}, \alpha)$ and $\tau'(\vec{\alpha})$ that $\Delta_{\tau[\tau'/\alpha]}(\vec{r}) = \Delta_\tau(\vec{r}, \Delta_{\tau'}(\vec{r}))$, for any $\vec{r}$. This follows easily from the definition in Fig. 5, by induction on the structure of τ. Finally, the term-substitutivity property in Fig. 4 is an easy consequence of Definition 4.12 together with the previously established fact that the relation (10) is reflexive. □

The following lemma will help us to compare the logical relation with observational congruence.

Lemma 4.14*(i) If $M =_{\mathrm{obs}} M' : \tau$, then for any $S \in Term^\top(\tau)$, $S \top M$ holds if and only if $S \top M'$ does.*

(ii) Suppose $r \in Rel(\tau, \tau')$ is $\top\top$-closed. Then

$$M_1 =_{\mathrm{obs}} M_2 : \tau \;\&\; (M_1, M_1') \in r \;\&\; M_1' =_{\mathrm{obs}} M_2' : \tau' \;\Rightarrow\; (M_2, M_2') \in r.$$

(iii) Δ is adequate (cf. Definition 3.1(iii)).

Proof. Recall that by definition $=_{\mathrm{obs}}$ is the largest compatible, substitutive, and adequate relation. Note that the compatibility properties of $=_{\mathrm{obs}}$ imply that $S@M =_{\mathrm{obs}} S@M' : \_list$ holds if $M =_{\mathrm{obs}} M' : \tau$ does. Therefore the

adequacy of $=_{\mathrm{obs}}$ together with Theorem 4.6 give (i). Part (ii) follows from (i) and the assumption that $r = r^{\top\top}$.

For part (iii), note that by Theorem 4.13(ii), for each $\tau \in \mathit{Typ}$ we have that $(\mathit{Id}, \mathit{Id}) \in (\Delta_{\tau\ \mathit{list}})^{\top}$. Thus if $\emptyset \vdash M \Delta M' : \tau\ \mathit{list}$, i.e. if $(M, M') \in \Delta_{\tau\ \mathit{list}}$, then $\mathit{Id} \top M \Leftrightarrow \mathit{Id} \top M'$, and hence by Theorem 4.6, $M \Downarrow \mathbf{nil}_\tau \Leftrightarrow M' \Downarrow \mathbf{nil}_\tau$, as required for adequacy. □

Theorem 4.15 *Given $\Gamma \vdash M : \tau$ and $\Gamma \vdash M' : \tau$, M and M' are observationally congruent if and only if they are logically related:*

$$\Gamma \vdash M =_{\mathrm{obs}} M' : \tau \ \Leftrightarrow\ \Gamma \vdash M \,\Delta\, M' : \tau. \tag{11}$$

Proof. Combining Lemma 4.11(i), Definition 4.12 and Lemma 4.14(ii), we have

$$\Gamma \vdash M_1 =_{\mathrm{obs}} M_2 : \tau \ \&\ \Gamma \vdash M_1 \,\Delta\, M_1' : \tau \ \&\ \Gamma \vdash M_1' =_{\mathrm{obs}} M_2' : \tau' \ \Rightarrow \\ \Gamma \vdash M_2 \,\Delta\, M_2' : \tau.$$

Since $=_{\mathrm{obs}}$ and Δ are reflexive (the former by construction, the latter by Theorem 4.13), we can take $M_1 = M_1' = M_2 = M$ and $M_2 = M'$ in (12) to deduce the left-to-right implication in (11).

For the converse implication, first note that the compatibility and substitutivity properties of Δ (Theorem 4.13) imply that the equivalence relation it generates, $(\Delta \cup \Delta^{op})^*$, is a PCF^+ congruence relation (cf. the proof of Theorem 3.2). Moreover, $(\Delta \cup \Delta^{op})^*$ is adequate, because Δ is (Lemma 4.14(iii)). Therefore $(\Delta \cup \Delta^{op})^*$ is contained in the largest adequate congruence relation, $=_{\mathrm{obs}}$, and hence so is Δ. □

5 Applications of Theorem 4.15

We give two kinds of application: some general results about PCF^+ observational congruence (such as a 'ciu' theorem and extensionality properties) and some properties of particular PCF^+ types up to $=_{\mathrm{obs}}$ (Examples 3.3–3.5).

5.1 A PCF^+ 'ciu' theorem

Let us begin with a version of the 'closed instantiations of uses' (ciu) theorem of Mason and Talcott [22]. It is convenient to split this into two parts, to do with 'instantiations' and with 'uses' respectively. The first part reduces observational congruence of open terms to that of closed terms (of closed type), via closed substitution instances (which for PCF^+ involves substitutions both of types and terms). Then the second part permits us to check the observational congruence of two closed terms by considering their termination behaviour just in evaluation contexts (of list type). Here we will replace evaluation contexts by the equivalent notion of frame stacks (Definition 4.4) and use the characterisation of termination given by Theorem 4.6.

Theorem 5.1 (*PCF^+* **'ciu' theorem**) *For each closed type $\tau \in Typ$, define a binary relation on $Term(\tau)$ by*

$$M =_{\mathrm{ciu}} M' : \tau \stackrel{\mathrm{def}}{\Leftrightarrow} \forall S \in Stack(\tau, \_\, list)\,(S \top M \;\Leftrightarrow\; S \top M'). \tag{12}$$

Then given $\Gamma \vdash M : \tau$ and $\Gamma \vdash M' : \tau$, with $\Gamma = \alpha_1, \ldots, \alpha_m, x : \tau_1, \ldots, x : \tau_n$ say, we write

$$\Gamma \vdash M =_{\mathrm{ciu}} M' : \tau$$

to mean that for all $\sigma_i \in Typ$ $(i = 1, \ldots, m)$ and all $N_j \in Term(\tau_j[\vec{\sigma}/\vec{\alpha}])$ $(j = 1, \ldots, n)$, it is the case that $M[\vec{\sigma}/\vec{\alpha}, \vec{N}/\vec{x}] =_{\mathrm{ciu}} M'[\vec{\sigma}/\vec{\alpha}, \vec{N}/\vec{x}] : \tau[\vec{\sigma}/\vec{\alpha}]$. Then $=_{\mathrm{ciu}}$ coincides with PCF^+ observational congruence:

$$\Gamma \vdash M =_{\mathrm{ciu}} M' : \tau \;\Leftrightarrow\; \Gamma \vdash M =_{\mathrm{obs}} M' : \tau.$$

Proof. The fact that $=_{\mathrm{obs}}$ is contained in $=_{\mathrm{ciu}}$ follows immediately from the fact that $=_{\mathrm{obs}}$ is, by definition, an adequate PCF^+ congruence relation.

For the converse implication, by Theorem 4.15, it suffices to show that $=_{\mathrm{ciu}}$ is contained in Δ. But it is evident from (12) that any $\top\top$-closed term-relation respects $=_{\mathrm{ciu}}$ and hence (by Definition 4.12) that

$$\Gamma \vdash M_1 =_{\mathrm{ciu}} M_2 : \tau \;\&\; \Gamma \vdash M_1 \,\Delta\, M_1' : \tau \;\&\; \Gamma \vdash M_1' =_{\mathrm{ciu}} M_2' : \tau' \;\Rightarrow\; \Gamma \vdash M_2 \,\Delta\, M_2' : \tau.$$

Since it is clear from its definition that $=_{\mathrm{ciu}}$ is reflexive, and since Δ is reflexive by Theorem 4.13, we can take $M_1 = M_1' = M_2 = M$ and $M_2 = M'$ to deduce that

$$\Gamma \vdash M =_{\mathrm{ciu}} M' : \tau \;\Rightarrow\; \Gamma \vdash M \,\Delta\, M' : \tau$$

as required. □

Fig. 8 gives some basic properties of $=_{\mathrm{obs}}$ (for simplicity, stated just for closed terms). All except (vii) are more or less immediate consequences of Theorem 5.1. Property (vii) gives a coinductive characterisation of observational congruence of lazy lists (cf. [24], for example). It follows by combining Theorem 4.15 with Remark 4.10. An example of its use occurs in the proof of Example 3.5 (see Sec. 5.5).

We turn next to the proofs of the properties of *null*, *unit*, and *list* types claimed in Examples 3.3–3.5.

5.2 Proof of Example 3.3

Suppose G is a closed term of type $null \stackrel{\mathrm{def}}{=} \forall \alpha\,(\alpha)$. We have to show that $G =_{\mathrm{obs}} \Omega : null$, where $\Omega \stackrel{\mathrm{def}}{=} \Lambda \alpha\,(\mathbf{fix}(\lambda x : \alpha\,(x)))$.

By properties (ii) and (vi) in Fig. 8, it suffices to show for all $\tau \in Typ$ that $G\tau =_{\mathrm{obs}} \mathbf{fix}(\lambda x : \tau\,(x)) : \tau$. For this it suffices, by Theorem 5.1, to show for all $S \in Term^{\top}(\tau)$ that $S \top (G\tau)$ does not hold, because evaluation of $\mathbf{fix}(\lambda x : \tau\,(x))$ does not converge.

Beta-conversions

(i) $(\lambda x : \tau_1 \,(M))\, A \;=_{\mathrm{obs}}\; M[A/x] : \tau_2$

(ii) $(\Lambda \alpha \,(M))\, \tau_2 \;=_{\mathrm{obs}}\; M[\tau_2/\alpha] : \tau_1[\tau_2/\alpha]$

(iii) $(\mathbf{case}\ \mathbf{nil}_{\tau_1}\ \mathbf{of\ nil} \Rightarrow M_1 \mid h :: t \Rightarrow M_2) \;=_{\mathrm{obs}}\; M_1 : \tau_2$
$(\mathbf{case}\ H :: T\ \mathbf{of\ nil} \Rightarrow M_1 \mid h :: t \Rightarrow M_2) \;=_{\mathrm{obs}}\; M_2[H/h, T/t] : \tau_2.$

(iv) $\mathbf{fix}(F) \;=_{\mathrm{obs}}\; F\,\mathbf{fix}(F) : \tau.$

Extensionality properties

(v) $F =_{\mathrm{obs}} F' : \tau_1 \to \tau_2$ if and only if for all $A \in \mathit{Term}(\tau_1)$, $F\,A =_{\mathrm{obs}} F'\,A : \tau_2$.

(vi) $G =_{\mathrm{obs}} G' \in \forall \alpha \,(\tau_1)$ if and only if for all $\tau_2 \in \mathit{Typ}$, $G\,\tau_2 =_{\mathrm{obs}} G'\,\tau_2 : \tau_1[\tau_2/\alpha]$.

(vii) $L =_{\mathrm{obs}} L' : \tau\ \mathit{list}$ if and only if $(L, L') \in r$ for some $r \in \mathit{Rel}(\tau\ \mathit{list}, \tau\ \mathit{list})$ satisfying that whenever $(M, M') \in r$ then
- $M \Downarrow \mathbf{nil}_\tau$ if and only if $M' \Downarrow \mathbf{nil}_\tau$
- if $M \Downarrow H :: T$, then $L' \Downarrow H' :: T'$ for some H' and T' with $H =_{\mathrm{obs}} H' : \tau$ and $(T, T') \in r$
- if $M' \Downarrow H' :: T'$, then $L \Downarrow H :: T$ for some H and T with $H =_{\mathrm{obs}} H' : \tau$ and $(T, T') \in r$.

Eta-conversions

(viii) $F =_{\mathrm{obs}} \lambda x : \tau \,(F\,x) : \tau_1 \to \tau_2$ (where $x \notin \mathit{fv}(F)$).

(ix) $G =_{\mathrm{obs}} \Lambda \alpha \,(G\,\alpha) : \forall \alpha \,(\tau)$ (where $\alpha \notin \mathit{ftv}(G)$).

(x) $\Omega\,(\tau_1 \to \tau_2) =_{\mathrm{obs}} \lambda x : \tau_1 \,(\Omega\,\tau_2)$ (where $\Omega \stackrel{\mathrm{def}}{=} \Lambda \alpha \,(\mathbf{fix}(\lambda x : \alpha \,(x)))$).

(xi) $\Omega\,(\forall \alpha \,(\tau)) =_{\mathrm{obs}} \Lambda \alpha \,(\Omega\,\tau)$.

Fig. 8. Some basic properties of PCF^+ observational congruence

From Theorem 4.13(ii) we have $(G, G) \in \Delta_{\forall \alpha \,(\alpha)} = \forall r \,(r^{\top\top})$. In other words, for all $\tau, \tau' \in \mathit{Typ}$ and $r \in \mathit{Rel}(\tau, \tau')$ we have

$$(G\,\tau, G\,\tau') \in r^{\top\top}. \tag{13}$$

Given τ, we use (13) with $\tau' = \tau\ \mathit{list}$ and r the one-element term-relation

$$r \stackrel{\mathrm{def}}{=} \{(\Omega\,\tau,\ \Omega\,(\tau\ \mathit{list}))\}.$$

For any $S \in \mathit{Term}^{\top}(\tau)$, let $S' \in \mathit{Term}^{\top}(\tau\ \mathit{list})$ be a frame stack that diverges when applied to any term of type $\tau\ \mathit{list}$, say

$$S' \stackrel{\mathrm{def}}{=} \mathit{Id} \circ (\mathbf{case}\ -\ \mathbf{of\ nil} \Rightarrow \Omega\,(\tau\ \mathit{list}) \mid h :: t \Rightarrow \Omega\,(\tau\ \mathit{list})).$$

Now neither $S \top (\Omega\,\tau)$ nor $S' \top (\Omega\,(\tau\ \mathit{list}))$ hold, because of the divergence properties of Ω. Therefore by definition of r, we have $(S, S') \in r^{\top}$. Combining this with (13) yields $S \top (G\,\tau) \Leftrightarrow S' \top (G\,\tau\ \mathit{list})$. But S' was chosen so that $S' \top L$ does not hold for any $L \in \mathit{Term}(\tau\ \mathit{list})$. Therefore $S \top (G\,\tau)$ does not hold either, as required. □

5.3 A graph lemma

In order to prove Examples 3.4 and 3.5 we use the following source of $\top\top$-closed term-relations.

Lemma 5.2 (Graphs of frame stacks are $\top\top$-closed)
For each $S \in Stack(\tau, \tau')$ the term-relation $graph_S \in Rel(\tau, \tau')$ defined by

$$graph_S \stackrel{\text{def}}{=} \{(M, M') \mid S@M =_{\text{obs}} M' : \tau'\}.$$

is $\top\top$-closed. (The definition of the application operation $-@-$ was given just after Definition 4.5.)

Proof. We have to show that $(graph_S)^{\top\top} \subseteq graph_S$. Note that by Theorem 4.15

$$graph_S = \{(M, M') \mid (S@M, M') \in \Delta_{\tau'}\}. \tag{14}$$

If $S' \circ S$ denotes the result of appending the frames in S to a frame stack S', then an induction on the length of S yields

$$(S' \circ S) \top M \Leftrightarrow S' \top (S@M). \tag{15}$$

From (14) and (15) we get

$$(S', S'') \in (\Delta_{\tau'})^{\top} \Rightarrow (S' \circ S, S'') \in (graph_S)^{\top}$$

and hence that

$$(N, N') \in (graph_S)^{\top\top} \Rightarrow (S@N, N') \in (\Delta_{\tau'})^{\top\top}.$$

But by Lemma 4.11(i), $(\Delta_{\tau'})^{\top\top} = \Delta_{\tau'}$. Therefore if $(N, N') \in (graph_S)^{\top\top}$, then $(S@N, N') \in \Delta_{\tau'}$ and hence $(N, N') \in graph_S$, as required. □

5.4 Proof of Example 3.4

Suppose G is a closed term of type $unit \stackrel{\text{def}}{=} \forall\alpha\,(\alpha \to \alpha)$. In view of the properties of $=_{\text{obs}}$ given in Fig. 8, to establish the claim in this example it suffices to show for all $\tau \in Typ$ and $M \in Term(\tau)$ that either

$$G\,\tau\,M =_{\text{obs}} \Omega\,\tau : \tau \tag{16}$$

or

$$G\,\tau\,M =_{\text{obs}} M : \tau. \tag{17}$$

Given $\tau \in Typ$ and $M \in Term(\tau)$, let $S \in Stack(unit\ list, \tau)$ be the frame stack

$$S \stackrel{\text{def}}{=} Id \circ (\mathbf{case}\ -\ \mathbf{of\ nil} \Rightarrow M \mid h :: t \Rightarrow M)$$

and consider $graph_S \in Rel(unit\ list, \tau)$ as in Lemma 5.2. By Theorem 4.13(ii) we have $(G, G) \in \Delta_{unit} = \forall r\,(r^{\top\top} \to r^{\top\top})$. So since by Lemma 5.2 $graph_S$ is $\top\top$-closed, we have

$$(G\ unit\ list, G\,\tau) \in graph_S \to graph_S. \tag{18}$$

By property (iii) in Fig. 8, we have $(\mathbf{nil}_{unit}, M) \in graph_S$. Therefore from (18) we get that $(G\ unit\ list\ \mathbf{nil}_{unit}, G\,\tau\,M) \in graph_S$, i.e. that

$$\mathbf{case}\,(G\ unit\ list\ \mathbf{nil}_{unit})\ \mathbf{of\ nil} \Rightarrow M \mid h :: t \Rightarrow M =_{\text{obs}} G\,\tau\,M : \tau. \tag{19}$$

Now either $G\ unit\ list\ \mathbf{nil}_{unit} \Downarrow C$ for some C, or not. In the first case we get

$$\mathbf{case}\,(G\ unit\ list\ \mathbf{nil}_{unit})\ \mathbf{of\ nil} \Rightarrow M \mid h :: t \Rightarrow M =_{\text{ciu}} M : \tau$$

and in the second we get

$$\mathbf{case}\,(G\ \mathit{unit\ list}\ \mathbf{nil}_{\mathit{unit}})\ \mathbf{of\ nil} \Rightarrow M \mid h :: t \Rightarrow M\ =_{\mathrm{ciu}}\ \Omega\tau : \tau.$$

Then by Theorem 5.1 and (19), the first possibility yields (17), whereas the second yields (16). □

5.5 Proof of Example 3.5

Let $L(\alpha)$, I and J be as defined in Example 3.5. By the results in Sec. 5.1, to prove

$$\alpha,\ \ell : \alpha\ \mathit{list} \vdash J\,\alpha\,(I\,\alpha\,\ell) =_{\mathrm{obs}} \ell : \alpha\ \mathit{list}$$
$$\alpha,\ g : L(\alpha) \vdash I\,\alpha\,(J\,\alpha\,g) =_{\mathrm{obs}} g : L(\alpha)$$

it suffices to show for all $\tau \in \mathit{Typ}$, $L \in \mathit{Term}(\tau\ \mathit{list})$, and $G \in \mathit{Term}(L(\tau))$ that

$$J\,\tau\,(I\,\tau\,L)\ =_{\mathrm{obs}}\ L : \tau\ \mathit{list} \tag{20}$$

and

$$I\,\tau\,(J\,\tau\,G)\ =_{\mathrm{obs}}\ G : L(\tau).$$

For the latter, in view of the definition of $L(\tau)$ it suffices to show for all $\tau' \in \mathit{Typ}$, $M' \in \mathit{Term}(\tau')$, and $F \in \mathit{Term}(\tau \to \tau' \to \tau')$ that

$$I\,\tau\,(J\,\tau\,G)\,\tau'\,M'\,F\ =_{\mathrm{obs}}\ G\,\tau'\,M'\,F : \tau'. \tag{21}$$

We tackle (20) first. Applying the beta-conversion properties in Fig. 8 to the definitions of I and J yields

$$I\,\tau\,L\,\tau'\,M'\,F\ =_{\mathrm{obs}}\ \mathbf{case}\,L\,\mathbf{of\ nil} \Rightarrow M' \mid h :: t \Rightarrow F\,h\,(I\,\tau\,t\,\tau'\,M'\,F) : \tau\ \mathit{list} \tag{22}$$

(for all L, M', and F of appropriate type) and then

$$J\,\tau\,(I\,\tau\,L)\ =_{\mathrm{obs}}\ \mathbf{case}\,L\,\mathbf{of\ nil} \Rightarrow \mathbf{nil}_{\tau} \mid h :: t \Rightarrow h :: (J\,\tau\,(I\,\tau\,t)) : \tau\ \mathit{list}. \tag{23}$$

From (23) it follows that $r \stackrel{\mathrm{def}}{=} \{(M, M') \mid M =_{\mathrm{obs}} J\,\tau\,(I\,\tau\,M') : \tau\ \mathit{list}\}$ satisfies the bisimulation conditions in property (vii) of Fig. 8 and hence is contained in $=_{\mathrm{obs}}$. Since $(J\,\tau\,(I\,\tau\,L), L) \in r$, we have (20).

Turning to the proof of (21), consider the frame stack $S \in \mathit{Stack}(\tau\ \mathit{list}, \tau')$ defined by

$$S\ \stackrel{\mathrm{def}}{=}\ \mathit{Id} \circ (\mathbf{case}\ -\ \mathbf{of\ nil} \Rightarrow M' \mid h :: t \Rightarrow (F\,h)\,(I\,\tau\,t\,\tau'\,M'\,F)).$$

In view of (22), we have $S@L =_{\mathrm{obs}} I\,\tau\,L\,\tau'\,M'\,F : \tau\ \mathit{list}$ and therefore

$$r_{M',F} \stackrel{\mathrm{def}}{=} \{(L, M'') \mid I\,\tau\,L\,\tau'\,M'\,F =_{\mathrm{obs}} M'' : \tau\ \mathit{list}\}$$

is a TT-closed member of $\mathit{Rel}(\tau\ \mathit{list}, \tau')$ by Lemma 5.2. So for each $G \in \mathit{Term}(L(\tau))$, since

$$(G, G) \in \Delta_{L(\tau)} = \forall r\,(r^{\mathrm{TT}} \to (\Delta_{\tau} \to r^{\mathrm{TT}} \to r^{\mathrm{TT}}) \to r^{\mathrm{TT}})$$

we have that

$$(G\,\tau\ \mathit{list}, G\,\tau') \in r_{M',F} \to (\Delta_{\tau} \to r_{M',F} \to r_{M',F}) \to r_{M',F}.$$

Vanilla PCF
(call-by-name evaluation; termination at function types is not observable)

$$\Delta_{\tau_1\to\tau_2}(\vec{r}) \stackrel{\text{def}}{=} \Delta_{\tau_1}(\vec{r}) \to \Delta_{\tau_2}(\vec{r})$$

where in general

$$r_1 \to r_2 \stackrel{\text{def}}{=} \{(F,F') \mid \forall\,(A,A') \in r_1\,((F\,A, F'\,A') \in r_2)\}.$$

'Lazy' PCF
(call-by-name evaluation; termination at function types is observable)

$$\Delta_{\tau_1\to\tau_2}(\vec{r}) \stackrel{\text{def}}{=} (\lambda\,\Delta_{\tau_1}(\vec{r})\,(\Delta_{\tau_2}(\vec{r})))^{\top\top}$$

where in general

$$\lambda r_1\,(r_2) \stackrel{\text{def}}{=} \{(\lambda x:\tau_1\,(M), \lambda x:\tau_1\,(M')) \mid \forall\,(A,A') \in r_1\,((M[A/x], M'[A'/x]) \in r_2)\}.$$

Call-by-value PCF
(call-by-value evaluation; hence termination at function types is necessarily observable)

$$\Delta_{\tau_1\to\tau_2}(\vec{r}) \stackrel{\text{def}}{=} (\lambda_v\,\Delta_{\tau_1}(\vec{r})\,(\Delta_{\tau_2}(\vec{r})))^{\top\top}$$

where in general

$$\lambda_v r_1\,(r_2) \stackrel{\text{def}}{=} \{(\lambda x:\tau_1\,(M), \lambda x:\tau_1\,(M')) \mid \forall\,(C,C') \in r_1 \text{ with } C,C' \text{ canonical}\,((M[C/x], M'[C'/x]) \in r_2)\}.$$

Fig. 9. Some actions of $\to$ on term-relations

From (22) and the definition of $r_{M',F}$ we get that $N \stackrel{\text{def}}{=} \Lambda\,\alpha\,(\mathbf{nil}_\alpha)$ and $C \stackrel{\text{def}}{=} \Lambda\,\alpha\,(\lambda\,h:\alpha\,(\lambda\,t:\alpha\;\mathit{list}\,(h :: t)))$ satisfy

$$(N\,\tau, M') \in r_{M',F} \qquad \text{and} \qquad (C\,\tau, F) \in \Delta_\tau \to r_{M',F} \to r_{M',F}$$

and hence

$$(G\,\tau\;\mathit{list}\,(N\,\tau)\,(C\,\tau)\,,\; G\,\tau'\,M'\,F) \in r_{M',F}.$$

Therefore $I\,\tau(G\,\tau\;\mathit{list}\,(N\,\tau)\,(C\,\tau))\,\tau'\,M'\,F =_{\text{obs}} G\,\tau'\,M'\,F : \tau'$, from which (21) follows by definition of J. □

6 Conclusion

Notions of contextual equivalence of programs have a final, as opposed to initial, character—in that program phrases are identified as much as possible within some observational framework. Therefore it is reasonable to expect $\forall$-types to have strong parametricity properties with respect to such a notion of equivalence. The unpublished work of Mitchell and Moggi on the maximally consistent model of PLC vindicates this expectation, and the work presented here provides further evidence, this time in a context more directly relevant to functional programming. It seems that in the presence of fixpoints, poly-

morphic types have very rich properties up to contextual equivalence and that operationally-based logical relations provide a convenient way of proving these properties. The applications in the previous section are certainly just a small selection of the results which can be proved using the machinery of Sec. 4. The Galois connection $(-)^\top$ between term-relations and stack-relations (Definition 4.7) seems the most interesting ingredient of that machinery. One of its roles is to tie the operational semantics into the logical relation. This idea is reinforced in Fig. 9, where we mention some alternative actions of $\to$ on term-relations (cf. Definition 4.2) which fit contextual equivalence for 'lazy' and call-by-value PCF. (Of course in each case, the definition of $- \top -$ and hence of $(-)^{\top\top}$, changes to match the changed operational semantics and/or observational scenario; and in the second case the notion of frame stack is different as well.)

As mentioned in Sec. 4, another role of the $(-)^\top$ operation is to provide a syntactic version of the domain-theoretic notion of admissibility. The recent upsurge in operational techniques in the semantics of higher order programming languages has been fuelled to a certain extent by developing syntactical versions of domain-theoretic methods (see [21,4] for example). Here it may be interesting to go in the opposite direction. The Galois connection $(-)^\top$ arose from purely operational considerations (in fact, as a way of dealing with dynamic allocation of local state in the logical relation introduced in [26]); but it may be useful to use a denotational version of $(-)^\top$ for 'extensional collapses' when constructing models of polymorphism and recursion. Denotationally, strict continuous functions play the role of frame stacks (evaluation contexts). So given domains D and D', we may consider the evident Galois connection between relations $R \subseteq D \times D'$ and relations $S \subseteq (D \multimap I) \times (D' \multimap I)$ induced by

$$f \top d \stackrel{\text{def}}{\Leftrightarrow} f(d) = \top$$

where $I = \{\bot, \top\}$ is the two-element domain with $\bot \sqsubseteq \top$ and $D \multimap I$ denotes the usual domain of strict continuous functions from D to I. It would be interesting, and possibly useful, to have a more explicit characterisation of what are the $\top\top$-closed relations in this sense. [4]

The particular type system of PCF^+ was chosen merely to be able to state and prove Example 3.5. I believe that the techniques presented in this paper will extend quite smoothly to show operational isomorphisms between appropriate pure PLC types and other type-theoretic notions important to programming language theory, such as recursive types. One direction which will be pursued in future work is the combination of $\forall$-types with intuitionistic linear types (! and $\multimap$ types). Plotkin [29] has pointed out that in the presence of fixpoints, and with relational parametricity, this system provides a

[4] One thing is clear: such a $\top\top$-closed relation on $D \times D'$ is in general something more than just a chain-closed and bottom-containing relation; thanks to Glynn Winskel (private communication) for pointing this out.

very expressive denotational metalanguage. Equipping the type system with a suitable operational semantics and associated notion of observational congruence, techniques similar to the ones introduced here should provide a term-model construction for the formal version of parametricity that Plotkin had in mind for this system. Another interesting direction (from the point of view of the foundations of object-oriented programming) would be to define operationally-based logical relations for combinations of subtyping, existential polymorphism, and recursion (cf. [31]).

References

[1] M. Abadi, L. Cardelli, and P.-L. Curien. Formal parametric polymorphism. *Theoretical Computer Science*, 121:9–58, 1993.

[2] M. Abadi and G. D. Plotkin. A per model of polymorphism and recursive types. In *5th Annual Symposium on Logic in Computer Science*, pages 355–365. IEEE Computer Society Press, Washington, 1990.

[3] E. S. Bainbridge, P. J. Freyd, A. Scedrov, and P. J. Scott. Functorial polymorphism. *Theoretical Computer Science*, 70:35–64, 1990. Corrigendum in 71:431, 1990.

[4] L. Birkedal and R. Harper. Relational interpretation of recursive types in an operational setting (Summary). In *Proc. TACS'97*, Lecture Notes in Computer Science. Springer-Verlag, Berlin, 1997. To appear.

[5] C. Böhm and A. Berarducci. Automatic synthesis of typed λ-programs on term algebras. *Theoretical Computer Science*, 39:135–154, 1985.

[6] L. Cardelli. Type systems. In *CRC Handbook of Computer Science and Engineering*, chapter 103, pages 2208–2236. CRC Press, 1997.

[7] T. Coquand, C. A. Gunter, and G. Winskel. DI-domains as a model of polymorphism. In M. Main, A. Melton, M. Mislove, and D. Schmidt, editors, *Mathematical Foundations of Programming Language Semantics*, volume 298 of *Lecture Notes in Computer Science*, pages 344–363. Springer-Verlag, Berlin, April 1987.

[8] T. Coquand, C. A. Gunter, and G. Winskel. Domain theoretic models of polymorphism. *Information and Computation*, 81:123–167, 1989.

[9] M. Felleisen and R. Hieb. The revised report on the syntactic theories of sequential control and state. *Theoretical Computer Science*, 103:235–271, 1992.

[10] J.-Y. Girard. *Interprétation fonctionelle et élimination des coupures dans l'arithmetique d'ordre supérieur.* PhD thesis, Université Paris VII, 1972. Thèse de doctorat d'état.

[11] J.-Y. Girard. *Proofs and Types.* Cambridge University Press, 1989. Translated and with appendices by Y. Lafont and P. Taylor.

[12] A. D. Gordon. Operational equivalences for untyped and polymorphic object calculi. In Gordon and Pitts [13], pages 9–54.

[13] A. D. Gordon and A. M. Pitts, editors. *Higher Order Operational Techniques in Semantics.* Publications of the Newton Institute. Cambridge University Press, 1998.

[14] R. Harper and M. Lillibridge. Explicit polymorphism and CPS conversion. In *20th SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 206–219. ACM Press, January 1993.

[15] R. Harper and C. Stone. A type-theoretic account of Standard ML 1996 (version 2). Technical Report CMU–CS–96–136R, Carnegie Mellon University, Pittsburgh, PA, September 1996.

[16] R. Hasegawa. Parametricity of extensionally collapsed term models of polymorphism and their categorical properties. In T. Ito and A. R. Meyer, editors, *Theoretical Aspects of Computer Software*, volume 526 of *Lecture Notes in Computer Science*, pages 495–512. Springer-Verlag, Berlin, 1991.

[17] R. Hasegawa. Categorical data types in parametric polymorphism. *Mathematical Structures in Computer Science*, 4:71–110, 1994.

[18] S. B. Lassen. Relational reasoning about contexts. In Gordon and Pitts [13], pages 91–135.

[19] S. B. Lassen. *Relational Reasoning about Functions and Nondeterminism.* PhD thesis, Department of Computer Science, University of Aarhus, 1998.

[20] Q. Ma and J. C. Reynolds. Types, abstraction, and parametric polymorphism, part 2. In S. Brookes, M. Main, A. Melton, M. Mislove, and D. A. Schmidt, editors, *Mathematical Foundations of Programming Semantics, Proceedings 1991*, volume 598 of *Lecture Notes in Computer Science.* Springer-Verlag, Berlin, 1992.

[21] I. A. Mason, S. F. Smith, and C. L. Talcott. From operational semantics to domain theory. *Information and Computation*, 128(1):26–47, 1996.

[22] I. A. Mason and C. L. Talcott. Equivalence in functional languages with effects. *Journal of Functional Programming*, 1:287–327, 1991.

[23] G. Morrisett, D. Walker, K. Crary, and N. Glew. From System F to typed assembly language. In *25rd SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages ?–? ACM Press, January 1998.

[24] A. M. Pitts. Operationally-based theories of program equivalence. In P. Dybjer and A. M. Pitts, editors, *Semantics and Logics of Computation*, Publications of the Newton Institute, pages 241–298. Cambridge University Press, 1997.

[25] A. M. Pitts. Reasoning about local variables with operationally-based logical relations. In P. W. O'Hearn and R. D. Tennent, editors, *Algol-Like Languages*, volume 2, chapter 17, pages 173–193. Birkhauser, 1997. First appeared in *Proceedings 11th Annual IEEE Symposium on Logic in Computer Science*, Brunswick, NJ, July 1996, pp 152–163.

[26] A. M. Pitts and I. D. B. Stark. Operational reasoning for functions with local state. In Gordon and Pitts [13], pages 227–273.

[27] G. D. Plotkin. Lambda-definability and logical relations. Memorandum SAI-RM-4, School of Artificial Intelligence, University of Edinburgh, October 1973.

[28] G. D. Plotkin. LCF considered as a programming language. *Theoretical Computer Science*, 5:223–255, 1977.

[29] G. D. Plotkin. Second order type theory and recursion. Notes for a talk at the Scott Fest, February 1993.

[30] G. D. Plotkin and M. Abadi. A logic for parametric polymorphism. In M. Bezem and J. F. Groote, editors, *Typed Lambda Calculus and Applications*, volume 664 of *Lecture Notes in Computer Science*, pages 361–375. Springer-Verlag, 1993.

[31] G. D. Plotkin, M. Abadi, and L. Cardelli. Subtyping and parametricity. In *9th Annual Symposium on Logic in Computer Science*, pages 310–319. IEEE Computer Society Press, Washington, 1994.

[32] J. C. Reynolds. Towards a theory of type structure. In *Paris Colloquium on Programming*, volume 19 of *Lecture Notes in Computer Science*, pages 408–425. Springer-Verlag, Berlin, 1974.

[33] J. C. Reynolds. Types, abstraction and parametric polymorphism. In R. E. A. Mason, editor, *Information Processing 83*, pages 513–523. North-Holland, Amsterdam, 1983.

[34] J. C. Reynolds and G. D. Plotkin. On functors expressible in the polymorphic typed lambda calculus. *Information and Computation*, 105:1–29, 1993.

[35] P. Wadler. Theorems for free! In *Fourth International Conference on Functional Programming Languages and Computer Architecture*, London, UK, September 1989.

[36] J. B. Wells. Typability and type-checking in the second-order λ-calculus are equivalent and undecidable. In *Proceedings, 9th Annual IEEE Symposium on Logic in Computer Science*, pages 176–185, Paris, France, 1994. IEEE Computer Society Press.

Electronic Notes in Theoretical Computer Science 10 (1998)
URL: http://www.elsevier.nl/locate/entcs/volume10.html 22 pages

Operational Subsumption, an Ideal Model of Subtyping

Laurent Dami [1]

Centre Universitaire d'Informatique
Université de Genève
Genève, Switzerland

Abstract

In a previous paper we have defined a semantic preorder called *operational subsumption*, which compares terms according to their error generation behaviour. Here we apply this abstract framework to a concrete language, namely the Abadi-Cardelli object calculus. Unlike most semantic studies of objects, which deal with typed equalities and therefore require explicitly typed languages, we start here from a untyped world. Type inference is introduced in a second step, together with an ideal model of types and subtyping. We show how this approach flexibly accommodates for several variants, and finally propose a novel semantic interpretation of structural subtyping as embedding-projection pairs.

1 Introduction

In a previous paper [10] we have defined a semantic preorder called *operational subsumption*, which compares terms according to their error generation behaviour. Together with the technical device of *labeled reductions*, used as a syntactic characterization of finite approximations, this semantics was shown to adequately interpret recursive types and subtyping. In this paper we apply this approach to **FOb**, the lambda-calculus of objects of Abadi and Cardelli [2]. Because we work with a concrete language instead of an abstract framework, several steps can be simplified, so the resulting semantic structure is intuitively quite obvious. Moreover, a "context lemma" in the untyped language gives us a simple induction principle for proving many properties of types. The goal is to show that the "Coverage of Operational Semantics" [21] can be widened to also deal with subtyping systems. More concretely, we

[1] work partially supported by Swiss SPP grant 5003-045332 and FNRS grant 2000-047181.96

show several directions where this approach can simplify or deepen previous results.

First, we give an interpretation of second-order bounded quantification for universal and existential types. This extends previous work on ideal models [18] with subtyping. Furthermore it answers to Abadi et al [3], who wondered whether their approach would apply to a suitable notion of operational ideals: this is exactly what is done here.

Then we have a direct way of interpreting typed equivalences of object-calculi (equivalences which depend on the type context in which objects are considered). Gordon and Rees [11] used the coinduction principle of Howe [14] to interpret these equivalences; however this required a heavy apparatus which switches between typed and untyped worlds: in addition to the reduction relation they had to define a labeled transition system and a "compatible refinement" relation. By contrast our interpretation is based on the untyped reduction relation. Moreover we can validate second-order typed equivalences, which remained an open issue in [11].

Finally we give a semantic interpretation of structural subtyping, which is useful for solving the problem known as "polymorphic object update". Bruce and Longo [7] have demonstrated that in usual interpretations of subtypes as subsets the polymorphic type $\forall X \leq T.X \to X$ can only contain the identity function, which makes it impossible to type some elementary updating operations on objects. The problem is developed in more detail in Chapter 16 of [2]. Some authors [13,20] have proposed to solve the problem by restricting the subtype relation in various ways so as to ensure that the subtypes have the same structure as the supertype. Here we show that usual subtyping and structural subtyping are two distinct notions semantically. The former corresponds to a subset relation, while the second corresponds to embedding-projection pairs in the subsumption ordering. Both subtyping notions can cohabit and could be included in the type syntax if so desired.

2 The untyped object calculus

The syntax shown in Figure 1 is built from the set ω of natural numbers, from a countable set $\mathcal{N}$ of *names* (for object fields) and a set $\mathcal{X}$ of *variables*. ε is a constant for errors. The main difference from [2] is that terms are *labeled*, i.e. decorated at each subterm with a natural number or ∞. Here the purpose of labels is merely to introduce a notion of finite projection for interpreting recursive types. In [10] we also used labels for an abstract definition of erroneous terms; this is not needed here, since we work in a concrete calculus for which the erroneous terms are just those which reduce to the error constant. Intuitively, each label acts as a counter limiting the number of interaction steps between the corresponding subterm and its context. When a label reaches 0, it becomes a divergent term, with which no interaction is possible. The infinite label ∞ imposes no limit, so for better readability it will usually be

$$
\begin{array}{lll}
\text{(indexes)} & i,j,k \in \omega & \\
\text{(labels)} & n,m \in \omega \cup \{\infty\} & \\
\text{(names)} & l,l' \in \mathcal{N} & \\
\text{(variables)} & x,y,z \in \mathcal{X} & \\
\text{(terms)} & a,b \in \mathcal{T} & ::= \; x^n \mid \varepsilon^n \mid a^n \mid (\lambda x.a)^n \mid (a\,b)^n \mid \\
 & & \quad [l_i = \varsigma x.a_i{}^{i\in I}]^n \mid (a.l)^n \mid (a \Leftarrow l = \varsigma x.b)^n \\
\text{(hnf)} & h \in \mathcal{H} & ::= \; x^{n+1} \mid (h\,a)^{n+1} \mid (h.l)^{n+1} \mid (h \Leftarrow l = \varsigma x.b)^{n+1} \\
\text{(values)} & v \in \mathcal{V} & ::= \; h \mid (\lambda x.a)^{n+1} \mid ([l_i = \varsigma x.a_i{}^{i\in I}])^{n+1} \mid \varepsilon^{n+1}
\end{array}
$$

Fig. 1. Syntax

omitted. In consequence there is an obvious embedding of usual, unlabeled terms into labeled terms by decorating each subterm with ∞. Furthermore ∞ is considered the successor of itself, so by abuse of notation a superscript $n+1$ may denote ∞, in which case n also equals ∞.

Like in the lazy λ-calculus, every function or object is a value if its label is > 0; furthermore open terms in head normal form (i.e. starting with a free variable) are also values. Finally, notice that ε is a value, which is a bit uncommon, but is an essential point of the approach.

We adopt common conventions for simplifying notation: $\lambda xy.a$ for $\lambda x.\lambda y.a$, $(a\,b\,c)$ for $((a\,b)c)$. In an object $[l_i = \varsigma x.a_i{}^{i\in I}]$ it is implicitly understood that the order of methods is irrelevant, and that for $i,j \in I, l_i \neq l_j$ whenever $i \neq j$. A similar convention will be used for types in Section 4. A method $l = a$ in an object is an abbreviation for $l = \varsigma x.a$, where x does not occur free in a; in that case it is called a *field*. Some common terms are $\mathbf{I} = \lambda x.x, \mathbf{K} = \lambda xy.y, \Omega = (\lambda x.xx)(\lambda x.xx)$. A *context* $C[-]$ is a term possibly containing occurrences of a "hole"; $C[a]$ is the term obtained by filling the holes with a, with possible variable capture. A *substitution* σ is a finite map from variables to terms; $a\sigma$ is the term obtained by substituting free occurrences of x in a by $\sigma(x)$, while avoiding variable capture; a single substitution is written $a[x := b]$. The sets $\mathcal{T}^c$ and $\mathcal{V}^c$ are respectively the closed terms and the closed values. A *closing substitution* for a is a σ such that $a\sigma \in \mathcal{T}^c$. The set $\mathcal{T}^n$ is the set of terms with outermost label less or equal to n.

The one-step reduction relation $\rightarrow$ is the least relation satisfying the rules in Figure 2. Labels and errors are the two unusual factors in these rules. Labels are decremented at each step where a term is "deconstructed"; in case $n+1 = n = \infty$, i.e. when the counter is infinite, the rules just become the usual rules for reduction of functions and objects. Errors are a way to avoid

$$
\begin{array}{ll}
(\lambda\beta) & (\lambda x.a)^{n+1}b \to (a[x := b])^n \\
(\lambda\sigma) & (\lambda x.a)^{n+1}.l \to \varepsilon^n \\
(\lambda\upsilon) & (\lambda x.a)^{n+1} \Leftarrow l = \varsigma x.b \to \varepsilon^n \\
(o\sigma) & [l_i = \varsigma x.a_i{}^{i\in I}]^{n+1}.l_j \to \\
 & \qquad \begin{cases} (a_j[x := [l_i = \varsigma x.a_i{}^{i\in I}]^{n+1}])^n & \text{if } j \in I \\ \varepsilon^n & \text{otherwise} \end{cases} \\
(o\upsilon) & [l_i = \varsigma x.a_i{}^{i\in I}]^{n+1} \Leftarrow l_j = \varsigma x.b \to [l_i = \varsigma x.a_i{}^{i\in I\setminus\{j\}}, l_j = \varsigma x.b]^n \\
(o\beta) & [l_i = \varsigma x.a_i{}^{i\in I}]^{n+1}\, b \to \varepsilon^n \\
(\varepsilon\beta) & (\varepsilon^{n+1}\, a) \to \varepsilon^n \\
(\varepsilon\sigma) & \varepsilon^{n+1}.l \to \varepsilon^n \\
(\varepsilon\upsilon) & \varepsilon^{n+1} \Leftarrow l = \varsigma x.b \to \varepsilon^n \\
(\lambda\varepsilon) & (\lambda x.\varepsilon^m)^{n+1} \to \varepsilon^{1+\min(m,n)} \\
(\nu^0) & a^0 \to \Omega \\
(\nu) & (a^m)^n \to a^{\min(m,n)} \\
\text{(cong)} & a \to b \implies \forall C[-], C[a] \to C[b]
\end{array}
$$

Fig. 2. Reduction rules

so-called "stuck terms" in the literature: instead of having terms which do not reduce but are not values, we explicitly reduce them to the error constant. Once generated, errors are always propagated further in the computation, i.e. there is no exception handling construct; however, since this is a call-by-name calculus, a context may discard an error in the same way that it would discard a divergent subterm: for example $\mathbf{K}\varepsilon$ reduces to $\mathbf{I}$.

The $(\lambda\varepsilon)$ rule is an ad hoc rule which allows us to greatly simplify the abstract framework of [10]: instead of observing "ability to interact" we will just observe reduction to ε. Intuitively the rule is motivated by the fact that a function containing ε can do nothing "useful" and therefore is equivalent to ε. By contrast, there is no such rule for objects, because a method containing ε can always be overridden.

In this untyped calculus the $\Leftarrow$ operator can not only override existing methods, but also add new methods, which is more liberal than in [2]. This is a deliberate choice, so that the same calculus can be used to interpret various type systems. In the next section we start with the type system of [2], in which only override can be well-typed; later we extend it with the system of [17] which also supports method extension.

The k-transitive closure of $\rightarrow$ is written $\xrightarrow{k}$, its reflexive, transitive closure is written $\xrightarrow{*}$, and the symmetric closure of $\xrightarrow{*}$ is written $\overset{*}{=}$.

Theorem 2.1 (Confluence) *The language is confluent: whenever $a \rightarrow b$ and $a \rightarrow c$ there is a d such that $b \xrightarrow{*} d$ and $c \xrightarrow{*} d$.*

Proof. Standard Tait technique using parallel reductions; see for example [22,9]. □

Definition 2.2 [Convergence]

A term a *converges* ($a \Downarrow$) iff $\exists v \in \mathcal{V}, a \xrightarrow{*} v$. Otherwise a *diverges* ($a \Uparrow$).

3 Operational Subsumption

The idea of operational subsumption is a simulation relation based on observation of errors. In the abstract framework of [10] we had to build a complex machinery in order to define the notion of "erroneous terms". Here this can be much simpler: like in [9], we have a rule ($\lambda\varepsilon$) which removes a λ-abstraction if its body is an error; this rule is admissible because it does not break confluence (Theorem 2.1 above). As a result it suffices to observe reductions to ε as a basis for subsumption.

Definition 3.1 [Error terms]

$$a\dagger \iff \exists n, a \xrightarrow{*} \varepsilon^{n+1}$$

$\mathcal{E}$ will denote the set $\{a \mid a\dagger\}$ of error terms.

Definition 3.2 [Contextual subsumption]

A term a *contextually subsumes* another term b, written $a \sqsubseteq^{ctxt}_{\varepsilon} b$, iff it generates fewer errors in all program contexts:

$$a \sqsubseteq^{ctxt}_{\varepsilon} b \iff \forall C[-], C[a]\dagger \implies C[b]\dagger$$

Subsumption is a lattice with bottom Ω and top ε. The symmetric closure of $\sqsubseteq_{\varepsilon}$ is written $\overset{\varepsilon}{=}$.

Lemma 3.3 *Subsumption contains reduction*

$$a \xrightarrow{*} b \implies a \overset{\varepsilon}{=} b$$

Proof. Direct from definition, knowing that the language is confluent. □

For convenience of proofs it is useful to establish a "context lemma" which allows us to only inspect a restricted set of contexts.

Definition 3.4 An *applicative context* $R[-]$ is a closed context generated by the following syntax:

$$R[-] \quad ::= \quad [-]^n \mid (R[-]\ a)^n \mid (R[-].l)^n \mid (R[-] \Leftarrow l = \varsigma x.a)^n$$

Definition 3.5 *Applicative subsumption* is the relation defined as

$$a \sqsubseteq^{appl}_{\varepsilon} b \iff \forall \sigma, \forall R[-], R[a\sigma]\dagger \implies R[b\sigma]\dagger$$

Lemma 3.6 (i) $\lambda x.a \sqsubseteq_{\varepsilon}^{appl} b \implies b\dagger \vee (b \xrightarrow{*} \lambda x.b' \wedge a \sqsubseteq_{\varepsilon}^{appl} b')$

(ii) $a \equiv [l_i = \varsigma x.a_i{}^{i\in I}] \sqsubseteq_{\varepsilon}^{appl} b \implies b\dagger \vee (b \xrightarrow{*} [l_j = \varsigma x.b_j{}^{j\in J}], J \subseteq I \wedge \forall j \in J, a_j[x := a] \sqsubseteq_{\varepsilon}^{appl} b_j[x := b])$

Proof.

(i) Since $(\lambda x.a).l\dagger$ and $((\lambda x.a) \Leftarrow l = \varsigma x.c)\dagger$, b cannot reduce to an object. So either $b\dagger$, or $b \xrightarrow{*} \lambda x.b'$. Then $R[a[x := c]] \sqsubseteq_{\varepsilon}^{appl} R[b'[x := c]]$ for every $R[-], c$, which implies $a \sqsubseteq_{\varepsilon}^{appl} b'$.

(ii) Similar reasoning.

□

Theorem 3.7 ("Ciu", context lemma)

$$a \sqsubseteq_{\varepsilon}^{ctxt} b \quad \iff \quad a \sqsubseteq_{\varepsilon}^{appl} b$$

Proof. The $\Rightarrow$ direction is trivial, since applicative contexts are contexts. The difficult part is the $\Leftarrow$ direction. We proceed by induction on the length of the proof of $C[a]\dagger$, i.e. we will show

$$\forall i, \forall C[-], ((C[a] \xrightarrow{i} \varepsilon^{n+1}) \wedge (a \sqsubseteq_{\varepsilon}^{appl} b)) \implies C[b]\dagger$$

The case $i = 0$ is trivial because then $C[-]$ is the empty context and both a and b are errors. If $i > 0$ and the first reduction step occurs either in $C[-]$ or in a, i.e. if $C[a] \to C'[a']$ with either $C[-] \to C'[-]$ or $a \to a'$, then we can directly use the induction hypothesis. Finally we can also ignore the cases where $b\dagger$, which again are trivial. So we are left with the following cases:

- cases $(\nu), (\nu^0)$: easy, a similar step can be performed with $C[b]$ and then we can appeal to the induction hypothesis.
- cases $(\varepsilon\beta), (\varepsilon\sigma), (\varepsilon\upsilon)$: easy again because both a and b must be error terms.
- cases $(\lambda\sigma), (\lambda\upsilon), (o\beta)$: these are the cases which generate an error. By the preceding Lemma a similar step can be performed with $C[b]$ and then the result follows from induction hypothesis.
- case $(\lambda\beta)$: here a must be of shape $\lambda x.a'$ and $C[-]$ must contain a subterm of shape $((-)B[-])$. Let $D[-], E[-]$ be the contexts repectively obtained by replacing this subterm by $(aB[-])$ or $(bB[-])$. Clearly $C[a] \equiv D[a]$ and $D[a] \to D'[a]$ where the same subterm is replaced by $a'[x := B[a]]$. We know $D'[a]\dagger$, and therefore by induction hypothesis $D'[b]\dagger$. Now by the preceding Lemma, $b \xrightarrow{*} \lambda x.b'$ with $a' \sqsubseteq_{\varepsilon}^{appl} b'$. Again by induction hypothesis, $E'[b]\dagger$, where $E'[b]$ is obtained from $D'[b]$ by replacing the same subterm by $b'[x := B[b]]$. Finally, since $E[b] \to E'[b]$ and $E[b] \equiv C[b]$, we have proved $C[b]\dagger$.
- cases $(o\sigma), (o\upsilon)$: like the preceding case, using the second clause of Lemma 3.6.

□

Because of this Theorem we will henceforth omit the ^{ctxt} or ^{appl} superscripts.

Example 3.8 Thanks to the preceding Theorem we can easily prove the following laws through induction on applicative contexts:

(i) $\lambda x_1 \ldots x_i.\varepsilon a_1 \ldots a_j \stackrel{\varepsilon}{=} \varepsilon$

(ii) $a \stackrel{\varepsilon}{\sqsubseteq} \lambda x.a\, x$ if x is not free in a

(iii) $[l_i = \varsigma x.a_i{}^{i \in I}] \stackrel{\varepsilon}{\sqsubseteq} [l_j = \varsigma x.a_j{}^{j \in J}]$ if $J \subseteq I$

(iv) $\neg([l = a, l' = \varsigma x.\, x.l] \stackrel{\varepsilon}{\sqsubseteq} [l' = a])$

(v) $[l = \varsigma x.[l = \varsigma y.x]] \stackrel{\varepsilon}{=} [l = \varsigma x.x]$

(vi) $[l_i = \varsigma x.a_i{}^{i \in I}, l = \varepsilon] \stackrel{\varepsilon}{=} [l_i = \varsigma x.a_i{}^{i \in I}]$

(vii) $[] \stackrel{\varepsilon}{\neq} \varepsilon$

The first law is quite obvious. The second law is the familiar η-rule of the λ-calculus; here it is not an equality because η-reduction is always sound, while η-expansion is obviously not sound when a is an object. The third law is the basis for object subtyping: an object with more methods subsumes an object with fewer methods. The fourth law shows that objects are compared not only on the basis of field access, but also on field update: accessing field l' would yield the same result on both sides, but updating field l would make the two object incomparable. The fifth law is an example that is not covered by the equational system of [2], because it cannot be shown by a finite proof; this is similar to the problem of equivalence or subtyping of recursive types [4]. The last two laws are consequences of our design choice to also support method addition in the untyped calculus; if, instead, we had only method override, then the situations would be reversed (law 6 would be an inequality and law 7 would be an equality).

4 Inference of Abadi-Cardelli types

This section introduces a hybrid type system, inspired from both [2] and [18], which includes object types, bounded universal and existential quantification, and recursive types. Like in [18], this is an impredicative, implicitly typed system, so there are neither type abstraction constructs for universal types nor **pack/open** constructs for existential types; such types are introduced and eliminated in the inference rules without help from the term syntax. Similary, the isomorphisms between recursive types and their unfoldings are handled automatically in the subtyping rules, so there is no need for explicit **fold/unfold** constructs in the term syntax. Apart from these surface differences, the typable objects are the same as in [2], so for example this system can type method override, but not method addition (a more powerful system will be discussed in the next section). On the other hand the only significant difference with respect to [18] is the addition of subtyping and bounded quantification. The intersection and union types of [18] are not handled here, not because of any technical difficulty, but just because they are of limited interest in the current context.

Type syntax

$$T, U ::= \text{TOP} \mid X \mid T \to U \mid [l_i : T_i^{i \in I}]$$
$$\mid \forall(X <: T)U \mid \exists(X <: T)U \mid \mu X.T$$

A *basis* Γ is a list of statements of the form $x : T$ (type assumption) or $X <: T$ (subtype assumption). A basis is well-formed iff the subjects of all assumptions are distinct, and the subject X of a subtype assumption $X <: T$ does not appear free in T nor in any previous assumption. *Judgements* are of the form $\Gamma \vdash T <: U$ (subtyping judgement) or $\Gamma \vdash a : T$ (typing judgement). The subtyping rules are given in Figure 3. These are the same as [2], except for the *fold/unfold* rules which express the isomorphism between recursive types. Observe that object types support width subtyping, but no depth subtyping (the types of the methods are invariant).

Type inference rules are defined in Figure 4. Most rules are standard. The rules for introduction and elimination of quantified types are taken from [18], with some adaptations for accommodating bounds on the quantified type variable.

The type interpretation is very similar to what we did in [10], except that here we have second-order types and object types instead of record types. Types are interpreted as ideals in $\langle \mathcal{T}^c, \sqsubseteq \rangle$, i.e. non-empty, downward-closed subsets of closed terms. Let **Tset** denote the set of such subsets. Letters t, u, v range over **Tset**. For any $t \in \mathbf{Tset}$, t^n denotes the set $\{a^n | a \in t\}$ (finite projection). **Tset** is a lattice ordered by subset inclusion, with top element $\top = \mathcal{T}^c$ and bottom element $\bot = \{a \in \mathcal{T}^c \mid a \Uparrow\}$. Notice that so far neither $\top$ nor $\bot$ has a corresponding expression in the type syntax. A *type environment* η is a mapping from type variables to **Tset**. Given a type environment, a type interpretation function $\mathbf{Ti}[-]$ maps types to members of **Tset**. Figure 5 gives the type interpretation. Like in [10], we interpret recursive types through indexed families of type interpretations, following an idea of [8]; non-contractive type expressions, like for example $\mu X.X$, are naturally mapped to the bottom type [2].

Lemma 4.1 (i) $\forall T, \eta, \mathbf{Ti}[T]_\eta \in \mathbf{Tset}$.

(ii) *A type is* trivial *if its interpretation contains* ε^1. *If* η *does not map any type variable to a trivial type, then* $\mathbf{Ti}[T]_\eta$ *is non-trivial for any* T.

Proof. Induction on T. □

Definition 4.2 A type environment η *satisfies* a basis Γ, written $\eta \models \Gamma$, iff $\mathbf{Ti}[X]_\eta \subseteq \mathbf{Ti}[T]_\eta$ whenever $(X <: T) \in \Gamma$. Similarly, a closing substitution on terms σ *satisfies* a basis Γ and a type environment η, written $\sigma \models (\Gamma, \eta)$, iff $x\sigma \in \mathbf{Ti}[T]_\eta$ whenever $(x : T) \in \Gamma$. The notation $\eta, \sigma \models \Gamma$ abbreviates $\eta \models \Gamma \wedge \sigma \models (\Gamma, \eta)$

[2] for an explanation of contractive, see the literature on recursive types, e.g. [8,4,18,2]

$$(top)\frac{}{\Gamma \vdash T <: \mathrm{TOP}}$$

$$(refl)\frac{}{\Gamma \vdash X <: X}$$

$$(env)\frac{}{\Gamma, X <: T \vdash X <: T}$$

$$(arrow)\frac{\Gamma \vdash T_2 <: T_1 \qquad \Gamma \vdash U_1 <: U_2}{\Gamma \vdash T_1 \to U_1 <: T_2 \to U_2}$$

$$(obj)\frac{}{\Gamma \vdash [l_i : T_i^{i \in I}, l_j : U_j^{j \in J}] <: [l_i : T_i^{i \in I}]}$$

$$(forall)\frac{\Gamma \vdash T_2 <: T_1 \qquad \Gamma, X <: T_2 \vdash U_1 <: U_2}{\Gamma \vdash \forall (X <: T_1) U_1 <: \forall (X <: T_2) U_2}$$

$$(exists)\frac{\Gamma \vdash T_1 <: T_2 \qquad \Gamma, X <: T_1 \vdash U_1 <: U_2}{\Gamma \vdash \exists (X <: T_1) U_1 <: \exists (X <: T_2) U_2}$$

$$(rec)\frac{\Gamma, X <: Y \vdash T <: U}{\Gamma \vdash \mu X.T <: \mu Y.U}$$

$$(unfold)\frac{}{\Gamma \vdash \mu X.T <: T[X := \mu X.T]}$$

$$(fold)\frac{}{\Gamma \vdash T[X := \mu X.T] <: \mu X.T}$$

Fig. 3. Subtyping rules

Theorem 4.3 (Type soundness)

(i) $\Gamma \vdash T <: U \implies \forall \eta \models \Gamma, \mathbf{Ti}[T]_\eta \subseteq \mathbf{Ti}[U]_\eta$.

(ii) $\Gamma \vdash a : T \implies \forall \eta, \sigma \models \Gamma, a\sigma \in \mathbf{Ti}[T]_\eta$.

(iii) $\Gamma \vdash a : T \implies \neg(a\dagger)$.

Proof. 1) induction on the judgement $\Gamma \vdash T <: U$; 2) induction on the judgement $\Gamma \vdash a : T$; 3) direct from 2) and from the preceding Lemma. □

$$(sub)\frac{\Gamma \vdash a : T \quad \Gamma \vdash T <: U}{\Gamma \vdash a : U}$$

$$(var)\frac{}{\Gamma, x : T \vdash x : T}$$

$$(lam)\frac{\Gamma, x : T \vdash a : U}{\Gamma \vdash \lambda x.a : T \to U}$$

$$(appl)\frac{\Gamma \vdash a : T \to U \qquad \Gamma \vdash b : T}{\Gamma \vdash (a\, b) : U}$$

$$(obj)\frac{\forall i : \Gamma, x : T \vdash a_i : T_i \quad (T \equiv [l_i : T_i^{i \in I}])}{\Gamma \vdash [l_i = \varsigma x.a_i^{i \in I}] : T}$$

$$(sel)\frac{\Gamma \vdash a : [l_i : T_i^{i \in I}]}{\forall i \in I : \Gamma \vdash a.l_i : T_i}$$

$$(upd)\frac{\Gamma \vdash a : T \quad \Gamma, x : T \vdash b : T_k \quad (T \equiv [l_i : T_i^{i \in I}]) \quad k \in I}{\Gamma \vdash a \Leftarrow l_k = \varsigma x.b : T}$$

$$(\forall intro)\frac{\Gamma, X <: T \vdash a : U}{\Gamma \vdash a : \forall (X <: T) U}$$

$$(\forall elim)\frac{\Gamma \vdash a : \forall (X <: U) T \qquad \Gamma \vdash V <: U}{\Gamma \vdash a : T[X := V]}$$

$$(\exists intro)\frac{\Gamma \vdash a : T[X := U] \qquad \Gamma \vdash U <: V}{\Gamma \vdash a : \exists (X <: V) T}$$

$$(\exists elim)\frac{\Gamma \vdash a : \exists (X <: T) U \qquad \Gamma, X <: T, x : U \vdash b : V}{\Gamma \vdash b[x := a] : V}$$

Fig. 4. Inference rules

5 Variations

This section explores some variants of the type system to augment its expressive power. Thanks to the underlying untyped languages and to Theorem 3.7 the soundness of the new rules can be checked quite easily.

$$\mathbf{Ti}[T]^0_\eta = \bot$$

$$\mathbf{Ti}[\mathrm{TOP}]^{n+1}_\eta = \{a \in \mathcal{T}^{n+1} | \neg(a\dagger)\}$$

$$\mathbf{Ti}[X]^{n+1}_\eta = \eta(X)^{n+1}$$

$$\mathbf{Ti}[T \to U]^{n+1}_\eta = \{a \in \mathcal{T}^{n+1} | b \in \mathbf{Ti}[T]^n_\eta \implies (a\,b) \in \mathbf{Ti}[U]^n_\eta\}$$

$$\mathbf{Ti}[[l_i : T_i^{i\in I}]]^{n+1}_\eta = \{a \in \mathcal{T}^{n+1} | a \sqsubseteq [] \wedge \forall i \in I, a.l_i \in \mathbf{Ti}[T_i]^n_\eta\}$$

$$\mathbf{Ti}[\forall(X <: T)U]^{n+1}_\eta = \bigcap_{t \subseteq \mathbf{Ti}[T]^{n+1}_\eta} \mathbf{Ti}[U]^{n+1}_{\eta[X \mapsto t]}$$

$$\mathbf{Ti}[\exists(X <: T)U]^{n+1}_\eta = \bigcup_{t \subseteq \mathbf{Ti}[T]^{n+1}_\eta} \mathbf{Ti}[U]^{n+1}_{\eta[X \mapsto t]}$$

$$\mathbf{Ti}[\mu X.T]^{n+1}_\eta = \mathbf{Ti}[T]^{n+1}_{\eta[X \mapsto \mathbf{Ti}[\mu X.T]^n_\eta]}$$

$$\mathbf{Ti}[T]_\eta = \{a | \forall n \in \omega, a^n \in \mathbf{Ti}[T]^n_\eta\}$$

Fig. 5. Type interpretation

5.1 Super-top type

The system presented above is a "classical" subtyping system in the sense that there is a maximal type TOP containing all non-erroneous terms. If we had union types, TOP could be expressed as the union of all function and object types:

$$\mathrm{TOP} = \mu X.(X \to X) \cup []$$

In [10] we have argued in favour of an even bigger type containing all terms, including erroneous ones. This can be easily incorporated into the system, by adding a new symbol $\top$ in the type syntax with interpretation $\mathbf{Ti}[\top]^{n+1}_\eta = (\mathcal{T}^c)^{n+1}$, and by adding the typing rule

$$(\top)\frac{}{\Gamma \vdash a : \top}$$

All previous results are preserved, except that for type soundness we have to exclude the set **Triv** of the *trivial* types generated by the following syntax:

$$Z \in \mathbf{Triv} ::= \top \mid T \to Z \mid \forall(X <: T)Z \mid \exists(X <: T)Z \mid \mu X.Z$$

Then the last point of Theorem 4.3 has to be reformulated as:

$$\Gamma \vdash a : T \wedge T \notin \mathbf{Triv} \implies \neg(a\dagger)$$

The interest of the $\top$ type is that the following subtyping rule is sound:

$$(obj\top)\frac{}{\Gamma \vdash [l_i : T_i^{i\in I}] <: [l_i : T_i^{i\in I}, l : \top]}$$

Hence a method with type $\top$ is equivalent to an absent method. We briefly repeat the example of [10] to show that this subtyping rule allows us to type more programs. Consider a translating function

$$\begin{aligned} T \stackrel{\text{def}}{=} \lambda x.[\,&\texttt{imprime} = x.\texttt{print},\\ &\texttt{affiche} = x.\texttt{display},\\ &\texttt{ferme} = x.\texttt{close}\ ] \end{aligned}$$

which takes an "english" object with three fields as argument, and returns a corresponding "french" object. Now consider object $O \stackrel{\text{def}}{=}$ [display = "hello"] and program $(T\ O).\texttt{affiche}$. This reduces to "hello" without error; however it cannot be typed in the original system because the type of T is

$$\begin{aligned} \forall X, Y, Z,\ &[\texttt{print} : X, \texttt{display} : Y, \texttt{close} : Z] \rightarrow\\ &[\texttt{imprime} : X, \texttt{affiche} : Y, \texttt{ferme} : Z] \end{aligned}$$

and [display : **String**] (the type of O) cannot be unified with the left-hand side of the arrow. By contrast, the $\top$ extension allows us to infer

$$\emptyset \vdash O : [\texttt{print} : \top, \texttt{display} : \textbf{String}, \texttt{close} : \top]$$

and therefore the program $(T\ O).\texttt{affiche}$ has type **String**.

5.2 Diamond types

Liquori [17] proposed a type system for the object calculus which supports method extension. This is done through so-called *diamond types* of shape

$$[l_i : T_i \diamond l_j : T_j]_{j\in J}^{i\in I} \quad (\{l_i | i \in I\} \cap \{l_j | j \in J\} = \emptyset)$$

where the left part of the diamond expresses the available methods as usual, while the right part specifies the safe possible method extensions. The main subtyping rules are displayed in Figure 6.

Liquori establishes soundness of his system through a subject reduction theorem, showing that types are preserved during computation. In our framework soundness can be shown by proving that the typing rules agree with the type interpretation. Since we have method extension in the underlying untyped calculus, the addition and verification of diamond types is direct. We

$$(Shift\diamond)\frac{}{\Gamma \vdash [l_i : T_i \diamond l_j : T_j]_{j\in J}^{i\in I+K} <: [l_i : T_i \diamond l_j : T_j]_{j\in J+K}^{i\in I}}$$

$$(Extend\diamond)\frac{}{\Gamma \vdash [l_i : T_i \diamond l_j : T_j]_{j\in J}^{i\in I} <: [l_i : T_i \diamond l_j : T_j]_{j\in J+K}^{i\in I}}$$

$$(Sat\diamond)\frac{}{\Gamma \vdash [l_i : T_i \diamond l_j : T_j]_{j\in J}^{i\in I} <: [l_i : T_i]^{i\in I}}$$

Fig. 6. Diamond subtyping

interpret diamond types as

$$\begin{aligned}&\mathbf{Ti}[[l_i : T_i^{i\in I} \diamond l_j : T_j^{j\in J}]]_\eta^{n+1} = \\ &\quad \{a \in \mathbf{Ti}[[l_i : T_i^{i\in I}]]_\eta^{n+1} \mid \forall l_j, \forall b, \\ &\qquad ((j \in J \wedge b[x := a] \in \mathbf{Ti}[T_j]_\eta^{n+1}) \vee j \notin J) \\ &\qquad\quad \Longrightarrow a{\Leftarrow}l_j = \varsigma x.b \in \mathbf{Ti}[[l_i : T_i^{i\in I\cup\{j\}}]]_\eta^n\}\end{aligned}$$

So this says that a method extension is safe if, whenever the field is mentioned in the right-hand side of the diamond, the body of the added method has a corresponding type. On the other hand if the field is not mentioned then there is no constraint on that field and the extension is safe in any case. Once this is understood the soundness of the subtyping rules for diamond types is easy to establish.

Lemma 5.1 *The subtyping rules of Figure 6 are sound.*

Proof. Omitted □

In addition we can easily verify other properties of diamond types, not mentioned in [17]. For example diamond types are contravariant on the right, i.e. the rule

$$(Contr\diamond)\frac{\forall j \in J, U_j <: T_j}{\Gamma \vdash [l_i : T_i \diamond l_j : T_j]_{j\in J}^{i\in I} <: [l_i : T_i \diamond l_j : U_j]_{j\in J}^{i\in I}}$$

is sound. Furthermore our "supertop" type $\top$ is compatible with diamond types and again is equivalent to absent fields, so the following rule is also sound:

$$(\top\diamond)\frac{}{\Gamma \vdash [l_i : T_i \diamond l_j : T_j]_{j\in J}^{i\in I} = [l_i : T_i, l : \top \diamond l_j : T_j, l' : \top]_{j\in J}^{i\in I}}$$

Because of lack of space we do not repeat here the type inference rules of [17]; however it should be clear that soundness of these rules can be established by similar means, i.e. without need for a subject-reduction theorem.

6 Typed equivalences

In calculi with subtyping, the equivalence relationship between terms is dependent on the type context: for example the objects $[l_1 = 2, l_2 = 4]$ and $[l_1 = \varsigma x.(x.l_3), l_2 = \texttt{"foo"}, l_3 = 2]$ are equal at type $[l_1 : \mathbf{Int}]$, because the only possible observations at this type are on field l_1, where the two objects have the same value. This is precisely why types and subtypes are usually interpreted in partial equivalence relationships (PERs), which express exactly this relation; however PERs require denotational domains to be built from. In this section we show how this can be done in our operational framework through a type indexing of the subsumption relation.

Definition 6.1 [Relevant contexts] A context $C[-]$ is *relevant* iff $(a \Uparrow \Longrightarrow C[a] \Uparrow)$ and $C[\varepsilon] \Downarrow$. The set of relevant contexts is written $\mathcal{C}$.

The idea here is that a context is irrelevant when no observation can be made about the term filling the hole. The first condition ensures that divergence at the hole is propagated to the outer level. The second condition rules out the contexts which are always divergent, because these also hide any observation from the hole.

Lemma 6.2

$$C[-] \in \mathcal{C} \implies C[\varepsilon]\dagger$$

Proof. By contradiction, showing that if $\neg(C[\varepsilon]\dagger)$ then $\neq (C[-] \in \mathcal{C})$. This is done by induction on the length of the reduction $C[\varepsilon] \xrightarrow{*} v$, comparing at each step with $C[\Omega]$. □

Definition 6.3 [Type-dependent Contexts]

$$\mathcal{C}_t \equiv \{C[-] \in \mathcal{C} | \forall a \in t, \neg(C[a]\dagger)\}$$

Lemma 6.4 (i) $\mathcal{C}_\perp = \mathcal{C}$

(ii) $\mathcal{C}_\top = \emptyset$

(iii) $\mathcal{C}_{\mathcal{T}\setminus\mathcal{E}} = \{C[-] | \forall a, \exists n, C[a] \xrightarrow{*} a^{n+1}\}$

(iv) $t \subseteq u \implies \mathcal{C}_u \subseteq \mathcal{C}_t$

Proof.

(i) for every relevant context $C[-]$ and divergent term $a \in \perp$, $C[a] \Uparrow$ and therefore $\neg(C[a]\dagger)$.

(ii) $\varepsilon \in \top$ and every relevant context filled with ε is erroneous (Lemma 6.2).

(iii) $\mathcal{T} \setminus \mathcal{E}$ contains both objects and functions. Therefore $C[-]$ cannot contain a subterm of shape $([-]B[-])$, because the hole could be filled by an object and the β reduction would yield an error. Conversely, the hole cannot either participate in a σ or υ reduction. Therefore the only reductions involving the hole must be ν reductions. Finally $C[-]$ cannot take a to a^0 because then it would be irrelevant.

(iv) If $C[-] \in \mathcal{C}_u$ then $\forall a \in u, \neg(C[a]\dagger)$; but then $\forall a \in t, \neg(C[a]\dagger)$ because $t \subseteq u$; therefore $C[-] \in \mathcal{C}_t$.

□

Definition 6.5 [Type-dependent subsumption]

$$a \mathrel{\underline{\sqsubseteq}}_t b \iff (a, b \in t \wedge (\forall C[-] \in \mathcal{C}_t, C[a]\dagger \implies C[b]\dagger))$$

Lemma 6.6 (i) $(\mathrel{\underline{\sqsubseteq}}_\perp) = (\mathrel{\underline{\sqsubseteq}})$

(ii) $(\mathrel{\underline{\sqsubseteq}}_\top) = \mathcal{T} \times \mathcal{T}$

(iii) $(\mathrel{\underline{\sqsubseteq}}_{\mathrm{TOP}}) = (\mathcal{T} \setminus \mathcal{E}) \times (\mathcal{T} \setminus \mathcal{E})$

(iv) $t \subseteq u \implies ((\mathrel{\underline{\sqsubseteq}}_t) \subseteq (\mathrel{\underline{\sqsubseteq}}_u))$

Proof. Direct consequences of Lemma 6.4. □

Now we overload the notation to define a family of subsumption relations indexed by syntactic types (as opposed to the semantic types used above).

Definition 6.7 [Syntactic type-dep. subsumption]

$$a \mathrel{\underline{\sqsubseteq}}^n_{T,\eta} b \iff a \mathrel{\underline{\sqsubseteq}}_{\mathbf{Ti}[T]^n_\eta} b$$

Lemma 6.8 (i) $\forall a, b \in \mathcal{T}^{n+1}, a \mathrel{\underline{\sqsubseteq}}^{n+1}_{\top,\eta} b$

(ii) $\forall a, b \in \mathcal{T}^{n+1} \setminus \mathcal{E}, a \mathrel{\underline{\sqsubseteq}}^{n+1}_{\mathrm{TOP},\eta} b$

(iii) $a \mathrel{\underline{\sqsubseteq}}^{n+1}_{T\to U,\eta} b \iff \forall c \in \mathbf{Ti}[T]^n_\eta, (ac) \mathrel{\underline{\sqsubseteq}}^n_{U,\eta} (bc)$

(iv) $Let\, T \equiv [l_i : T_i^{i\in I}]\,.\, a \mathrel{\underline{\sqsubseteq}}^{n+1}_{T,\eta} b \iff \forall i \in I,$

$a.l_i \mathrel{\underline{\sqsubseteq}}^n_{T_i,\eta} b.l_i \wedge$

$(c[x := b] \in \mathbf{Ti}[T_i]^{n+1}_\eta \implies$

$(a \Leftarrow l_i = \varsigma x.c) \mathrel{\underline{\sqsubseteq}}^n_{T,\eta} (b \Leftarrow l_i = \varsigma x.c))$

(v) $a \mathrel{\underline{\sqsubseteq}}^{n+1}_{\forall(X<:T)U,\eta} b \iff \forall t \subseteq \mathbf{Ti}[T]^{n+1}_\eta, a \mathrel{\underline{\sqsubseteq}}^{n+1}_{U,\eta[X\mapsto t]} b$

(vi) $a \mathrel{\underline{\sqsubseteq}}^{n+1}_{\exists(X<:T)U,\eta} b \iff \exists t \subseteq \mathbf{Ti}[T]^{n+1}_\eta, a \mathrel{\underline{\sqsubseteq}}^{n+1}_{U,\eta[X\mapsto t]} b$

(vii) $a \mathrel{\underline{\sqsubseteq}}^{n+1}_{\mu X.T,\eta} b \iff a \mathrel{\underline{\sqsubseteq}}^{n+1}_{T[X:=\mu X.T],\eta} b$

Proof. Double induction on the shape of types and on the index n. □

Definition 6.9 [Typed equalities] The interpretation of typed equalities is:

$$\mathbf{Ti}[\Gamma \vdash a \leftrightarrow b : T] \quad = \quad \forall \eta, \sigma \models \Gamma, \forall n, (a\sigma \mathrel{\overset{\varepsilon}{=}}^n_{T,\eta} b\sigma)$$

Notice that this interpretation accounts for open objects and open types.

Lack of space prevents us from checking the complete set of equational rules of [2]. However it is worth noticing that for most of them we can take advantage of Lemma 6.6: if we are able to prove $a \overset{\varepsilon}{=} b$ then $a \leftrightarrow b : T$ at any T. This in particular covers all "Eval" rules of [2], because $a \xrightarrow{*} b \implies a \overset{\varepsilon}{=} b$ (Property 3.3). So we will concentrate on a few interesting rules that are more specifically related to subtyping. These are displayed in Figure 7. The first

$$\text{(Eq Sub Obj)}\ \frac{T \equiv [l_i : T_i^{i\in I}] \quad T' \equiv [l_i : T_i^{i\in I\cup J}] \qquad \Gamma, x : T \vdash a_i : T_i \quad \forall i \in I \qquad \Gamma, x : T' \vdash a_j : T_j \quad \forall j \in J}{\Gamma \vdash [l_i = \varsigma x.a_i^{i\in I}] \leftrightarrow [l_i = \varsigma x.a_i^{i\in I\cup J}] : T}$$

$$\text{(Eq } \forall \text{ Intro)}\frac{\Gamma, X <: T \vdash a \leftrightarrow b : U}{\Gamma \vdash a \leftrightarrow b : \forall(X <: T)U}$$

$$\text{(Eq } \forall \text{ Elim)}\frac{\Gamma \vdash a \leftrightarrow b : \forall(X <: T)U \quad \Gamma \vdash T' <: T}{\Gamma \vdash a \leftrightarrow b : U[X := T']}$$

$$\text{(Eq } \exists \text{ Intro)}\frac{\Gamma \vdash a \leftrightarrow b : U[X := T'] \quad \Gamma \vdash T' <: T}{\Gamma \vdash a \leftrightarrow b : \exists(X <: T)U}$$

$$\text{(Eq } \exists \text{ Elim)}\frac{\Gamma \vdash a \leftrightarrow b : \exists(X <: T)U \quad \Gamma, X <: T, x : U \vdash c \leftrightarrow d : V}{\Gamma \vdash c[x := a] \leftrightarrow d[x := b] : V}$$

Fig. 7. Some equational rules

one is taken from [2]. The other rules have to do with second-order types and subtyping; similar rules can be found in the literature for explicitly typed systems (system $F_<$), but to the best of our knowledge the present formulation for implicitly typed systems is new.

Theorem 6.10 *The rules of Figure 7 are sound.*

Proof. (Eq Sub Obj): one direction comes directly from the untyped subsumption (Example 3.8, item 3). For the other direction, it suffices to observe that, for any η, contexts in $\mathcal{C}_{\mathbf{Ti}[T]_\eta}$ can only use names in $\{l_i | i \in I\}$. ($\forall$ and $\exists$ rules): first observe that $\forall\eta, \mathbf{Ti}[U[X := T]]_\eta = \mathbf{Ti}[U]_{\eta[X\mapsto \mathbf{Ti}[T]_\eta]}$; second, for any logical predicate $P(\eta)$ we have the following equivalence:

$$\forall\eta \models (\Gamma, X <: T).P(\eta) \equiv \forall\eta' \models \Gamma.\forall t \subseteq \mathbf{Ti}[T]_{\eta'}.P(\eta'[X \mapsto t])$$

Taking advantage of these facts, for example if we take $P(\eta) \equiv \forall n, a \stackrel{\varepsilon\, n}{=}_{U,\eta} b$ we immediately get a soundess proof for for (Eq $\forall$ Intro). Other rules are handled similarly. □

7 Structural subtyping

In the previous section, subtyping was interpreted as set inclusion. This is quite close in spirit to the two other interpretations found in the literature, namely coercion functions [6] or inclusions between partial equivalence relations (PERs) [7]. However it also suffers from the same problem, first explained in [7]: the bounded polymorphic type

$$\forall(X <: T)X \to X$$

can only contain the identity function. Intuitively this comes from the fact that for every member a of T there is a semantic subtype of T containing only that single member, and then the function can do nothing but map that member to itself. This is annoying because a function like

$$\lambda x.x \Leftarrow l = \varsigma y.\mathbf{not}(x.l)$$

cannot be assigned type

$$\forall X <: [l : \mathbf{Bool}].X \to X$$

Even if there is no syntactic type $[l : \mathbf{True}]$, there is an ideal $\{a \mid a.l \sqsubseteq^{\varepsilon} \mathbf{true}\}$ contained in **Bool** at which the specification $X \to X$ is unsound. In consequence there is no way to express in the type that this function leaves all fields other than l untouched.

Several approaches have been taken to fix the problem. As already noted in [7], the problem comes from the fact that the semantics contains "too many subtypes", typically many more than can be defined in the type syntax. One possibility then is to drop the denotational semantics altogether. Chapter 16 of [2] takes such an approach: it introduces stronger rules called "structural rules" which take advantage of the fact that all operations on objects preserve some implicit structure; these rules are unsound in the denotational semantics, but are proved to be operationally sound. Another possibility is to fix the semantic interpretation of subtyping. This program is carried out in [20], where subtyping is restricted to pointwise equality on fields between "record PERs"; however this interpretation prohibits depth subtyping on records, which is somewhat unsatisfactory because intuitively this form of subtyping is structural. Hofmann and Pierce [13] have a more general approach where the statement $T <: U$ is interpreted as a standard coercion $c : T \to U$ together with an overwrite function $put[T, U] : T \to U \to U$ which updates the "U part" of an element of T without changing the "remaining part". However this approach only works in cases of "positive subtyping", since there is no function dual to $put[T, U]$ to go down in the type hierarchy.

7.1 Structural subtyping as embedding-projections

Here we propose a new interpretation of subtyping by embedding-projection pairs, very similar in spirit to the maps used in domain theory for solving recursive equations through inverse limit constructions. The embedding $\uparrow_t^u : t \to u$ from a subtype to its supertype is just an inclusion map, so in the following there is no need to mention it explicitly. By contrast the projection $\downarrow_t^u : u \to t$ has to "preserve structure": the image must be equal to the argument at type u.

Definition 7.1 [structural inclusion]

$$t \subseteq\!\!\downarrow u \iff t \subseteq u \wedge \exists \downarrow_t^u : u \to t, \forall a \in u, \downarrow_t^u (a) = \min(\{b \in t | b \doteq_u^{\varepsilon} a\})$$

The essential condition here is that for every $a \in u$ there must be a $b \in t$

$$(struct - forall)\frac{\Gamma, X \lessdot\!\!\downarrow T \vdash U_1 \lessdot\!\!\downarrow U_2}{\Gamma \vdash \forall(X \lessdot\!\!\downarrow T)U_1 \lessdot\!\!\downarrow \forall(X \lessdot\!\!\downarrow T)U_2}$$

$$(struct - exists)\frac{\Gamma, X \lessdot\!\!\downarrow T \vdash U_1 \lessdot\!\!\downarrow U_2}{\Gamma \vdash \exists(X \lessdot\!\!\downarrow T)U_1 \lessdot\!\!\downarrow \exists(X \lessdot\!\!\downarrow T)U_2}$$

Fig. 8. Structural subtyping for quantified types

such that $b \stackrel{\epsilon}{=}_u a$. The canonical choice of the minimal element is a secondary condition which proves to be convenient for dealing with quantified types.

Before proceeding with our calculus it is worth considering informally why subtyping is not always structural. Consider examples such as $[1 \ldots 10] <: [1 \ldots 20]$ or **True** <: **Bool**. These make sense as subset inclusion, but the only way to map every element of the supertype to an element of the subtype is the constant function $\lambda x.\Omega$, which loses all information about its argument; therefore the "structure" of the supertype is lost. Similarly, the rules $T \cap U <: T$ or $\bot <: T$ in systems with intersection types or bottom types are non-structural. Finally, since universal and existential quantification are interpreted as intersection and union, the subtyping rules *forall* and *exists* in Figure 3 also break structure. Therefore it is not possible to just keep the system of Section 4 and reinterpret <: as $\subseteq\!\!\downarrow$.

A system involving *both* subtyping relations is perfectly conceivable: it suffices to add a distinction in the type syntax between ordinary subtyping statements $T <: U$ and structural statements $T \lessdot\!\!\downarrow U$. Then in addition to the ordinary quantified types we also would have structurally quantified types $\forall(X \lessdot\!\!\downarrow T)U$ and $\exists(X \lessdot\!\!\downarrow T)U$, with the obvious interpretation

$$\mathbf{Ti}[\forall(X \lessdot\!\!\downarrow T)U]^{n+1}_{\eta} = \bigcap_{t \subseteq\!\!\downarrow \mathbf{Ti}[T]^{n+1}_{\eta}} \mathbf{Ti}[U]^{n+1}_{\eta[X \mapsto t]}$$

$$\mathbf{Ti}[\exists(X \lessdot\!\!\downarrow T)U]^{n+1}_{\eta} = \bigcup_{t \subseteq\!\!\downarrow \mathbf{Ti}[T]^{n+1}_{\eta}} \mathbf{Ti}[U]^{n+1}_{\eta[X \mapsto t]}$$

However even if the semantic apparatus is ready, it is not clear whether having two distinct notions of subtyping simultaneously is a desirable feature. The advantage of structural subtyping is that it validates some more powerful typing rules, like the (*val structural update*) rule shown below. On the other hand it forbids some "atomic subtyping" rules such as **True** <: **Bool** or **Posint** <: **Int**. As discussed here, having both is possible, but at the cost of a greater complexity in the type system. Choosing the most appropriate solution is a matter of language design. Here we will briefly discuss a system with structural subtyping only.

Definition 7.2 [Structural subtyping] The system of structural subtyping defines judgements of shape $\Gamma \vdash T \lessdot\!\!\downarrow U$; it is obtained from the system of

Section 4 by

(i) replacing subtyping assumptions $X <: T$ in Γ by structural subtyping assumptions $X <\!\!\downarrow T$;

(ii) replacing $<:$ by $<\!\!\downarrow$ in the rules of Figure 3;

(iii) replacing rules (*forall*) and (*exists*) in Figure 3 by the weaker version given in Figure 8.

For this system we obviously reinterpret the statement $\eta \models \Gamma$ as $\mathbf{Ti}[X]_\eta \subseteq\!\!\downarrow \mathbf{Ti}[T]_\eta$ whenever $(X <\!\!\downarrow T) \in \Gamma$.

Theorem 7.3 (Soundness of structural subtyping)

$$\Gamma \vdash T <\!\!\downarrow U \implies \forall \eta \models \Gamma, \mathbf{Ti}[T]_\eta \subseteq\!\!\downarrow \mathbf{Ti}[U]_\eta$$

Proof. Induction on the proof of $\Gamma \vdash T <\!\!\downarrow U$. For each subtyping rule used we exhibit the witness function $\downarrow^{\mathbf{Ti}[U]_\eta}_{\mathbf{Ti}[T]_\eta}$, which will be abbreviated as $\downarrow^U_T$.

- case (*top*): $\downarrow^{TOP}_T = \lambda x.\Omega$.
- case (*refl*): $\downarrow^X_X = \lambda x.x$.
- case (*env*): from the assumption $\eta \models \Gamma, X <\!\!\downarrow T$ there exists a projection $\downarrow^{\mathbf{Ti}[T]_\eta}_{\mathbf{Ti}[X]_\eta}$.
- case (*arrow*): $\downarrow^{T_2 \to U_2}_{T_1 \to U_1} = \lambda f.\, \downarrow^{U_2}_{U_1} \circ f \circ \downarrow^{T_1}_{T_2}$.
- case (*obj*): $\downarrow^{[l_i:T_i^{i\in I}]}_{[l_i:T_i^{i\in I\cup J}]} = \lambda x.x \Leftarrow l_{j_1} = \Omega \ldots \Leftarrow l_{j_k} = \Omega \quad (J = \{j_1 \ldots j_k\})$
- case (*forall*): we know that $\forall \eta, \forall t \subseteq\!\!\downarrow \mathbf{Ti}[T]_\eta, \exists \downarrow^{\mathbf{Ti}[V]_{\eta[X\mapsto t]}}_{\mathbf{Ti}[U]_{\eta[X\mapsto t]}}$ Therefore we can build

$$\downarrow^{\forall(X<\!\!\downarrow T)V}_{\forall(X<\!\!\downarrow T)U} = \bigcap_{t \subseteq\!\!\downarrow \mathbf{Ti}[T]_\eta} \downarrow^{\mathbf{Ti}[V]_{\eta[X\mapsto t]}}_{\mathbf{Ti}[U]_{\eta[X\mapsto t]}}$$

  By the fact that each projection function canonically chooses the minimal element in its codomain, their intersection is well-defined and satisfies the conditions of Definition 7.1.
- case (*exists*): like the preceding case, taking $\bigcup$ instead of $\bigcap$.
- case (*rec*): using the premice $\Gamma, X <\!\!\downarrow Y \vdash T <\!\!\downarrow U$, we show by induction on n that the family of projections $\downarrow^{\mathbf{Ti}[\mu Y.U]^n_\eta}_{\mathbf{Ti}[\mu X.T]^n_\eta}$ is well-defined. Then the projection $\downarrow^{\mu Y.U}_{\mu X.T}$ is the union of such projections.
- cases $(fold), (unfold)$: $\downarrow^{\mu X.T}_{T[X:=\mu X.T]} = \downarrow^{T[X:=\mu X.T]}_{\mu X.T} = \lambda x.x$

□

Corollary 7.4 *The "val structural update" rule of [2] is sound:*

$$\frac{\Gamma \vdash a : T \quad \Gamma \vdash T <\!\!\downarrow [l_i : T_i^{i\in I}] \quad \Gamma, x : T \vdash b : T_j \quad j \in I}{\Gamma \vdash a \Leftarrow l_j = \varsigma x.b : T}$$

Proof. Suppose $\eta, \sigma \models \Gamma$ and $b\sigma[x := a] \in \mathbf{Ti}[T_j]_\eta$. Let $a' \equiv a \Leftarrow l_j = \varsigma x.b$ and $a'' \equiv \downarrow_T^{[l_i:T_i{}^{i\in I}]}(a')$: we must have $a'' \in \mathbf{Ti}[T]_\eta$ and $a'' \stackrel{\varepsilon}{=}_{[l_i:T_i{}^{i\in I}],\eta} a'$. Hence T is closed under updates of type T_j at l_j. □

Like [20], we now show an isomorphism which proves that type $\forall(X <\!\!\downarrow [l : T])X \to X$ contains more functions than just the identity.

Theorem 7.5 *The types* $\forall(X <\!\!\downarrow [l : T])X \to X$ *and* $T \to T$ *are isomorphic.*

Proof. Consider functions

$$F = \lambda f.\lambda x.(f([l = x])).l$$

$$G = \lambda f.\lambda x.x \Leftarrow l = f(x.l)$$

It is easy to see that $F \circ G = G \circ F = \mathbf{I}$, and that $F : (\forall(X <\!\!\downarrow [l : T])X \to X) \to (T \to T)$. By contrast structural subtyping is required for the reverse typing: $G : (T \to T) \to (\forall(X <\!\!\downarrow [l : T])X \to X)$ is only derivable if we have the *(val structural update)* rule above. □

8 Conclusion

Thanks to several recent works, operational techniques are regaining considerable interest. Results such as fixpoint induction, which once were only provable through denotational means, are now shown with operational bisimilarities [21]. However in these works subtyping was seldom taken into account. The framework proposed in [10], introducing an explicit error constant ε which becomes the top element of the semantic lattice, can remedy to this deficiency. By applying it here to the object calculus of Abadi and Cardelli, we have demonstrated that some complex aspects of the abstract framework (namely the defintion of erroneous terms) can be greatly simplified when working with a concrete calculus. Furthermore we have introduced a number of technical innovations or improvements to previous work:

- rules for bounded second-order types in an implicitly typed system;
- an operational interpretation of typed equalities which deals with open terms and open types;
- a semantic interpretation of "structural subtyping";
- a "super-top" type which improves the typing power of the system without impairing type soundness.

Acknowledgements

A previous draft of this paper received very detailed comments from anonyous referees; these were extremely useful to improve the quality of the paper. Comments from various participants to the HOOTS II workshop are also gratefully acknowledged.

References

[1] Samson Abramsky and C.-H. Luke Ong. Full Abstraction in the Lazy Lambda Calculus. *Information and Computation*, 105:159-267, 1993.

[2] Martin Abadi and Luca Cardelli. *A Theory of Objects.* Springer-Verlag, Monographs in Computer Science, 1996.

[3] Martin Abadi, Benjamin Pierce and Gordon Plotkin. Faithful Ideal Models for Recursive Polymorphic Types. *Int. J. of Foundations for Computer Science*, 2(1):1-21, 1991.

[4] Roberto Amadio and Luca Cardelli. Subtyping Recursive Types. *ACM Trans. on Prog. Lang and Systems*, 15(4):575-631, 1993.

[5] Henk Barendregt. *The Lambda-Calculus, its Syntax and Semantics.* Studies in Logic and the Foundations of Mathematics, North-Holland, 1984.

[6] Val Breazu-Tannen, Thierry Coquand, Carl A. Gunter, and Andre Scedrov. Inheritance as Implicit Coercion. *Information and Computation* 93:172-221, 1991. Also in [12], pp 197-245.

[7] Kim Bruce and Giuseppe Longo. A Modest Model of Records, Inheritance, and Bounded Quantification. *Information and Computation* 87:196-240, 1990. Also in [12], pp 151-195.

[8] Felice Cardone and Mario Coppo. Two extensions of Curry's Type Inference System. In *Logic and Computer Science*, P. Odifreddi(ed), pp 19-75. Academic Press, 1990.

[9] Laurent Dami. A Lambda-Calculus for Dynamic Binding. *Theoretical Comp. Sc.* 192(2):201-231, Feb 1998.

[10] Laurent Dami. Labeled Reductions, Runtime Errors, and Operational Subsumption. Extended abstract in *Proc. ICALP'97*, LNCS, Springer-Verlag, 1997. Expanded version in *Objects at Large*, Technical Report, University of Geneva, 1997, available from `http://cuiwww.unige.ch/~dami/publis.html`.

[11] Andrew Gordon and Gareth Rees. *Bisimilarity for a First-Order Calculus of Objects with Subtyping.* Technical Report 386, Computer Laboratory, University of Cambridge, January 1996, available at `http://www.cl.cam.ac.uk/ adg`. Technical summary in *Proceedings 23rd ACM POPL*, Jan 1996, pp 386-395.

[12] Carl A. Gunter and John C. Mitchell, eds. *Theoretical aspects of object-oriented programming: types,semantics, and language design.* MIT Press, Foundations of computing series, 1994.

[13] Martin Hofmann and Benjamin C. Pierce. Positive subtyping. *Information and Computation*, 126(1):11-33, 10 April 1996.

[14] D. J. Howe. Equality in lazy computation systems. In *Proc. 4th IEEE Symp. on Logic in Comp. Sc.*, pp 198-203, 1989.

[15] Trevor Jim and Albert R. Meyer. Full Abstraction and the Context Lemma. *SIAM J. on Computing* 25(3):663-696, June 1996.

[16] Marina Lenisa. Semantic Techniques for Deriving Coinductive Characterizations of Observational Equivalences for Lambda-Calculi. Proc. *TLCA'97*, pp 248-266. LNCS 1210, Springer-Verlag, 1997.

[17] Luigi Liquori. An Extended Theory of Primitive Objects: First Order System. *Proc. ECOOP'97*, pp 146-169, LNCS 1241, Springer-Verlag, 1997.

[18] David MacQueen, Gordon Plotkin and Ravi Sethi. An Ideal Model for Recursive Polymorphic Types. *Information and Control*, 71:95-130, 1986.

[19] Ian A. Mason, Scott F. Smith and Carolyn L. Talcott. From Operational Semantics to Domain Theory. *Information and Computation*, 128:26-47, 1996.

[20] Erik Poll. System F with Width-subtyping and Record Updating. *Proc. Internat. Symp. on Theoret. Aspects of Comp. Software*, Sendai, Japan. LNCS, Springer-Verlag, 1997.

[21] Scott Smith. The Coverage of Operational Semantics. In *Higher Order Operational Techniques in Semantics*, A. Gordon and A. Pitts, editors, Cambridge University Press, 1998.

[22] M. Takahashi. Parallel Reductions in λ-calculus. *Information and Computation*, 118(1):120-127, 1995.

Monadic Type Systems: Pure Type Systems for Impure Settings (Preliminary Report)

Gilles Barthe [1]

Chalmers Tekniska Högskola, Institutionen för Datavetenskap, Eklandagatan 86, Göteborg, S-412 96 Sweden, gillesb@cs.chalmers.se

John Hatcliff [2]

Oklahoma State University, Department of Computer Science, 219 Math Sciences, Stillwater, OK, USA, 74078, hatcliff@a.cs.okstate.edu

Peter Thiemann

Dept. of Computer Science, University of Nottingham, University Park, Nottingham NG7 2RD, England, pjt@cs.nott.ac.uk

Abstract

Pure type systems and *computational monads* are two parameterized frameworks that have proved to be quite useful in both theoretical and practical applications. We join the foundational concepts of both of these to obtain *monadic type systems.* Essentially, monadic type systems inherit the parameterized higher-order type structure of pure type systems and the monadic term and type structure used to capture computational effects in the theory of computational monads. We demonstrate that monadic type systems nicely characterize previous work and suggest how they can support several new theoretical and practical applications.

A technical foundation for monadic type systems is laid by recasting and scaling up the main results from pure type systems (confluence, subject reduction, strong normalisation for particular classes of systems, *etc.*) and from operational presentations of computational monads (notions of operational equivalence based on applicative similarity, co-induction proof techniques).

We demonstrate the use of monadic type systems with case studies of several call-by-value and call-by-name systems. Our framework allows to capture the restriction to value polymorphism in the type structure and is flexible enough to accommodate extensions of the type system, e.g., with higher-order polymorphism. The theoretical foundations make monadic type systems well-suited as a typed intermediate language for compilation and specialization of higher-order,

strict and non-strict functional programs. The monadic structure guarantees sound compile-time optimizations and the parameterized type structure guarantees sufficient expressiveness.

Key words: Functional programming, polymorphism, pure type systems, monads, operational semantics.

---

[1] Research supported by TMR Fellowship FMBICT972065.
[2] Research supported by NSF under grant CCR-9701418.

Contents

1 Introduction

The compilation of typed higher-order programming languages requires the use of intermediate languages. This is due to the large conceptual gap between source and target languages in such a translation. In addition, the use of an intermediate language yields a synergistic effect since common backends and program transformations may be shared among several source languages. Therefore, a large amount of flexibility in terms of operational properties as well as typing properties is required from an intermediate language.

Recent work in compilation has produced a number of compile-time frameworks that either stress the operational properties or the typing properties. But none of the frameworks considered so far offers parametricity in both respects. Our aim is to reconcile these hitherto separate concerns by joining in a single framework the fundamental concepts of pure type systems and those of computational monads.

Computational monads:

Moggi's theory of computational monads as embodied by his *computational metalanguage* parameterizes semantic specifications by a *notion of computation* (state transformation, I/O, *etc.*) [36]. Terms in the metalanguage can be reduced without regard to computational effects — even without specifying their nature — provided that they can be expressed with a monad. In addition, the metalanguage explicitly orders all operations with computational effects. A wide range of evaluation orders (call-by-name, call-by-value, and mixed strategies) can be encoded by choosing a suitable translation into the metalanguage. Thus, the metalanguage provides an *evaluation-order independent* framework for describing evaluation and reasoning in the presence of computational effects.

The computational metalanguage has been used successfully in a number of applications including compiling transformations, intermediate languages for partial evaluation, denotational foundations for languages with I/O, and structuring functional programs with effects (*e.g.*, [14, 16, 21, 27, 28, 32, 40, 50]). The benefits of the metalanguage are as follows.

Increased modularity: Compilation, partial evaluation, and semantics can be specified in terms of the metalanguage: all computational aspects of the source language are captured in the metalanguage; no extra information is necessary.

Correctness: The framework guarantees the soundness of compile-time reductions in the presence of computational effects.

Genericity: The metalanguage is general enough to encode the structure of a variety of source languages by choosing a suitable translation. The resulting translated programs are in *monadic normal form* [28] — a generalization of A-normal forms [16], where each intermediate result is named. This form is particularly suited to compiling.

However, most applications deal with untyped or simply-typed versions of the metalanguage and do not support parametricity of the type structure.

Pure type systems:

Pure type systems (PTS) [4] are a generic framework to define type systems and logics. Examples are F_2, F_ω, LF, and the Calculus of Constructions. They provide the theoretical foundation for proof assistants and modern programming language type systems. Subtle differences between the systems can be described by varying the *specification* upon which the framework is parameterized.

Compactness and parametricity make PTSs an attractive framework for typed intermediate languages [31]. This builds on a well-established practice of using subsystems of F_ω in compiling polymorphic programming languages like ML or Haskell [23,26,35,39,46,47]. It is by now an accepted fact that type information is beneficial in all phases of compilation and program transformation. PTSs are exceptionally well suited because of the following features.

Economy: Since terms and types are conflated in one syntactic category, the same functions can be used for their manipulation.

Extensibility: The intermediate language is robust with respect to changes to the type system of the source language: extensions to the type system usually amount to changing only the specification of the PTS.

Checkability Being explicitely typed, many PTSs have a decidable type-checking. Therefore all compile-time transformations can be type checked and thus debugged.

In many of these applications, PTSs (whose semantics are *free from computational effects*) are being used in settings where computational effects are ubiquitous. Whereas the PTS framework provides adequate treatment of the type structure, it does not ensure sound treatment of effects. This stems from the fact that β-conversion is the foundational notion of equality in PTSs. As a consequence, a PTS-based intermediate language (such as one might use for treating ML) must impose ad hoc restrictions to ensure soundness. Furthermore, it seems difficult to use a PTS-based language for languages that explicitly control the evaluation order. This situation arises in a Haskell compiler after strictness and/or totality analysis.

Monadic type systems:

Monadic type systems (MTSs) join the concepts of the computational metalanguage and of pure type systems. They inherit the parameterized higher-order type structure of pure type systems and the monadic term and type structure used to capture computational effects in the theory of computational monads. The combination shares the above-listed advantages of both frameworks—except for type-checking which is undecidable for languages that include dependent types—and provides some additional features. For example, an MTS specification can enforce the restriction to value polymorphism [34] by type checking the intermediate language. Furthermore, it is possible to encode other approaches to the sound treatment of effects in the presence of polymorphism (e.g., polymorphism by name [33]).

1.1 Contributions

This paper takes a first step towards combining the fundamental concepts of pure type systems and computational monads and towards developing operational theories for λ-calculi with dependent types.

- MTSs combine the fundamental concepts of pure type systems and computational monads, thus providing a typed intermediate framework that offers parametricity with respect to type disciplines and computational effects. [3]
- The main technical properties of PTSs (confluence, subject reduction, strong normalisation for particular classes of systems, *etc.*) are recast and proved for MTSs.
- The operational properties of the computational metalanguage (notions of operational equivalence based on applicative similarity, co-induction proof techniques) are recast and proved for MTSs where the notion of computation is *non-termination* as expressed by the *lifting monad.*
- A detailed case study (SML'97 with value polymorphism) demonstrates a specific instance of the MTS framework which captures the operational and typing properties that are desirable in a typed intermediate representation for SML'97. This particular language can serve as a basis for compiling ML and also for specialization of ML. It is noteworthy that specialization with respect to values and specialization with respect to types is indistinguishable in the language, which leads to some simplification inside a compiler/specializer. On the other hand, the monadic structure is indispensable to ensure the soundness of transformations inside a compiler/specializer.

1.2 Overview

The rest of the paper is organized as follows. Section 2 presents the basic definitions of monadic type systems. Section 3 provides intuition and illustrates how the MTS framework can be used to describe many systems and concepts appearing in the literature. Section 4 reviews technical properties of monadic type systems. Section 5 is devoted to the operational semantics of monadic type systems.

Section 6 assesses the applicability of MTS and compares them with related work. Section 7 gives conclusions and future work.

Throughout the paper, we assume some basic knowledge of computational monads and pure type systems.

2 Syntax of Monadic Type Systems

In PTSs, different typed λ-calculi are generated by varying PTS parameters called *specifications* that consist of *sorts*, *axioms* and *rules*. Sorts provide the type universes of the calculus. Axioms provide a "membership" relation between universes, while rules determine which function types may be formed and in which universe they live.

[3] Both pure type systems and the metalanguage have achieved a status of 'standards'. We do not claim that MTSs inherit this status and expect alternative or refined syntaxes to arise.

Logical Pure Type Systems [10] form a class of specifications that feature a distinguished sort $*$ of propositions or programs, that is required to comply with several requirements. Examples of Logical Pure Type Systems include systems in Barendregt's λ-cube [3, 4] which are obtained from specifications that use the sorts $*$ and $\Box$, with the axiom $* : \Box$. Intuitively, inhabitants of $*$ are types and inhabitants of types are programs, whereas inhabitants of $\Box$ are kinds and inhabitants of kinds are type constructors.

Monadic Type Systems are based on the same idea. The difference is that MTSs *also* include a fundamental concept from the theory of computational monads: they distinguish between types for *values* (completely evaluated terms, or terms with no computational effects) and types for *computations* (terms with remaining effectful computational steps). Technically, the distinction is achieved by distinguishing a sort of values and a sort of computations. As a result of our choice, the framework does not encompass every meaningful monadic type system—in the same way as the class of logical specifications does not encompass every meaningful pure type system—yet it allows for many interesting systems to be defined.

Definition 1 (specifications) A (MTS) *specification* is a tuple $\mathbf{S} = (\mathcal{S}, *^v, *^c, \mathcal{A}, \mathcal{R})$ where

- $\mathcal{S}$ is a set of *sorts*;
- $*^v \in \mathcal{S}$ is the sort of *values* and $*^c \in \mathcal{S}$ is the sort of *computations*;
- $\mathcal{A} \subseteq \mathcal{S} \times \mathcal{S}$ is a set of *axioms*;
- $\mathcal{R} \subseteq \mathcal{S} \times \mathcal{S} \times \mathcal{S}$ is a set of *product rules.*

The above Definition does not impose any requirement of the sorts, axioms or rules and thus allows for specifications that violate the monadic characteristics. These requirements (which yield what we call "strictly monadic specifications") are postponed until Section 4 and Section 5.

Monadic type systems have a single category of expressions, which are called pseudo-terms. Pseudo-terms are built from the standard constructors for pure type systems—including abstraction, application and dependent product—and for monads—including unit and let-expressions. In addition, pseudo-terms include a data type of natural numbers with some basic operations and a fixpoint construct that give a "PCF-like" presentation of monadic type systems. A more general presentation, which falls out of the scope of this paper, would be to include a scheme for inductive types, e.g. based on recent work by T. Coquand [9].

Definition 2 (pseudo-terms) The set $\mathcal{T}$ of *pseudo-terms* is defined by the abstract syntax

$$\begin{aligned}\mathcal{T} = {} & \mathcal{V} \mid \mathcal{S} \mid \mathcal{T}\,\mathcal{T} \mid \lambda\mathcal{V}{:}\mathcal{T}.\,\mathcal{T} \mid \Pi\mathcal{V}{:}\mathcal{T}.\,\mathcal{T} \mid \mathsf{let}\ \mathcal{V}{:}\mathcal{T} \Leftarrow \mathcal{T}\ \mathsf{in}\ \mathcal{T} \mid \mathsf{unit}\ \mathcal{T} \mid \mathsf{M}\ \mathcal{T} \mid \\ & \mathsf{fix}\ \mathcal{V}{:}\mathcal{T}.\,\mathcal{T} \mid \mathsf{nat} \mid \lceil n \rceil \mid \mathsf{succ}\ \mathcal{T} \mid \mathsf{pred}\ \mathcal{T} \mid \mathsf{if0}\ \mathcal{T}\ \mathcal{T}\ \mathcal{T}\end{aligned}$$

For technical convenience, we assume:

(i) variables to be *sorted*, i.e. $\mathcal{V} = \bigcup_{s \in \mathcal{S}} \mathcal{V}^s$ where the $\mathcal{V}^s$s are pairwise disjoint countably infinite sets;

(ii) Barendregt's variable conventions [2].

Substitution, free and bound variables, *etc.* are defined as usual. The notions of reduction for monadic type systems are drawn from those of pure type systems and those of the computational metalanguage.

Definition 3 (notions of reduction)

- β-reduction $\rightarrow_\beta$ is defined as the compatible closure of the rule

$$(\lambda x{:}A.\ M)\ N \rightarrow_\beta M\{x := N\}$$

- μ-reduction $\rightarrow_\mu$ is defined as the compatible closure of the rules

$$\mathsf{let}\ x{:}A \Leftarrow (\mathsf{unit}\ N)\ \mathsf{in}\ M \rightarrow_{\mu_1} M\{x := N\}$$
$$\mathsf{let}\ x{:}A \Leftarrow M\ \mathsf{in}\ (\mathsf{unit}\ x) \rightarrow_{\mu_2} M$$
$$\mathsf{let}\ x_2{:}A_2 \Leftarrow (\mathsf{let}\ x_1{:}A_1 \Leftarrow M_1\ \mathsf{in}\ M_2)\ \mathsf{in}\ M_3 \rightarrow_{\mu_3} \mathsf{let}\ x_1{:}A_1 \Leftarrow M_1\ \mathsf{in}\ (\mathsf{let}\ x_2{:}A_2 \Leftarrow M_2\ \mathsf{in}\ M_3)$$

  where in the last rule it is assumed that $x_2 \notin \mathrm{FV}(A_1) \cup \mathrm{FV}(M_1)$.

- ϕ-reduction $\rightarrow_\phi$ is defined as the compatible closure of the rule

$$\mathsf{fix}\ x{:}M.\ N \rightarrow_\phi N\{x := \mathsf{fix}\ x{:}M.\ N\}$$

- ι-reduction is defined as the compatible closure of the rule

$$\mathsf{if0}\ \lceil 0 \rceil\ M\ N \rightarrow_\iota M$$
$$\mathsf{if0}\ \lceil n+1 \rceil\ M\ N \rightarrow_\iota N$$
$$\mathsf{succ}\ \lceil n \rceil \rightarrow_\iota \lceil n+1 \rceil$$
$$\mathsf{pred}\ \lceil n+1 \rceil \rightarrow_\iota \lceil n \rceil$$
$$\mathsf{pred}\ \lceil 0 \rceil \rightarrow_\iota \lceil 0 \rceil$$
$$\mathsf{pred}\ (\mathsf{succ}\ x) \rightarrow_\iota x$$

- The notion of *ml*-reduction $\rightarrow_{ml}$ is defined as the union of β, ι, ϕ and μ-reductions.

We do not consider η-reduction since it is unsound for CPS translations and would complicate the meta-theory. The typing rules for MTSs are drawn from the rules of pure type systems and computational monads. In addition, there are rules for data types and fixpoints.

Definition 4 (derivability)

- A *pseudo-context* is a finite (ordered) list of pairs $x : A$ where $x \in \mathcal{V}$ and $A \in \mathcal{T}$. The set of pseudo-contexts is denoted by $\mathcal{C}$ and the empty context is denoted by $\langle\rangle$. The domain of a context Γ is

$$\mathrm{dom}(\Gamma) = \{x \mid \exists t \in \mathcal{T}.\ x : t \in \Gamma\}$$

  We use Γ, Δ as metavariables ranging $\mathcal{C}$.

Structural rules

$$\text{(axiom)} \quad \frac{(s_1, s_2) \in \mathcal{A}}{\langle\rangle \vdash s_1 : s_2}$$

$$\text{(conversion)} \quad \frac{\Gamma \vdash A : B \quad \Gamma \vdash B' : s \quad B =_{ml} B'}{\Gamma \vdash A : B'}$$

$$\text{(start)} \quad \frac{\Gamma \vdash A : s \quad x \in \mathcal{V}^s \setminus \mathrm{dom}(\Gamma)}{\Gamma, x : A \vdash x : A}$$

$$\text{(weakening)} \quad \frac{\Gamma \vdash A : B \quad \Gamma \vdash C : s \quad x \in \mathcal{V}^s \setminus \mathrm{dom}(\Gamma)}{\Gamma, x : C \vdash A : B}$$

Product rules

$$\text{(product)} \quad \frac{\Gamma \vdash A : s_1 \quad \Gamma, x : A \vdash B : s_2 \quad (s_1, s_2, s_3) \in \mathcal{R}}{\Gamma \vdash (\Pi x{:}A.\, B) : s_3}$$

$$\text{(abstraction)} \quad \frac{\Gamma, x : A \vdash b : B \quad \Gamma \vdash (\Pi x{:}A.\, B) : s}{\Gamma \vdash \lambda x{:}A.\, b : \Pi x{:}A.\, B}$$

$$\text{(application)} \quad \frac{\Gamma \vdash F : (\Pi x{:}A.\, B) \quad \Gamma \vdash a : A}{\Gamma \vdash F\, a : B\{x := a\}}$$

Fig. 1. Rules for Monadic Type Systems (structural and product rules)

- A *judgement* is a triple $\Gamma \vdash M : A$ where $\Gamma \in \mathcal{C}$ and $M, A \in \mathcal{T}$.
- The derivability relation $\vdash$ is induced by the rules of Figures 1 and 2. If $\Gamma \vdash M : A$ then Γ, M, A are *legal.*
- The MTS generated by $\mathbf{S}$ is the quadruple $\lambda\mathbf{S} = (\mathcal{T}, \mathcal{C}, \rightarrow_{ml}, \vdash)$.

The next Section illustrates the use of the framework through several instances of monadic type systems.

3 Instances of Monadic Type Systems

Each specification defines a particular instance of a monadic type system. In this section, we explore specifications that characterize the images of various translations that encode evaluation strategies in the computational metalanguage. The crucial point, in each case, is the distinction between values and computations. We illustrate this for some well-studied simply-typed languages, then—in more detail—for a fragment of ML, and finally for call-by-name and call-by-value versions of F_ω. The latter one is particularly interesting because it is quite close to the intermediate language used in the FLINT/ML compiler project [45].

All systems considered in this section are related to the λ-cube. They share the set of sorts and the axioms:

- $\mathcal{S} = \{*^v, *^c, \Box^v, \Box^c\}$;
- $\mathcal{A} = \{*^v : \Box^v, *^c : \Box^c\}$.

The intuition here is that the elements of $*^v$ are types of values, which are inhabited by effect-free terms, whereas the elements of $*^c$ are types of computations, which are inhabited by terms that may have computational effects: nontermination, in the most simple case. This distinction is used to enforce the value restriction of SML'97. Also certain operational properties of a term can be proved just from determining whether its type has kind $*^v$ or $*^c$. Similarly, the elements of $\Box^v$ are value kinds, which are inhabited by value type constructors, and likewise for $\Box^c$.

The rule sets of the specifications considered in this section do not involve $\Box^c$ at all. Abstractions involving $\Box^v$ are found to be sufficient, even to characterize the images of call-by-name translations, where we expect to perform abstraction over computations. Since we do not have a proof that $\Box^c$ and $\Box^v$ can be conflated to $\Box$, in general, we keep the more precise distinction for the time being.

3.1 Simply-typed systems

Many systems of interest only have product rules of the form $(s_1, s_2, *^v)$, which we write as $[s_1, s_2]$. They reflect the expected monadic typing of abstractions as values. Since simply-typed systems do not allow abstraction of types, the rules do not contain $\Box^v$ and $\Box^c$.

Monad rules

$$\text{(monad)}\quad \frac{\Gamma \vdash A : *^{x}}{\Gamma \vdash \mathsf{M}\ A : *^{c}}$$

$$\text{(unit)}\quad \frac{\Gamma \vdash M : A \quad \Gamma \vdash \mathsf{M}\ A : s}{\Gamma \vdash \mathsf{unit}\ M : \mathsf{M}\ A}$$

$$\text{(let)}\quad \frac{\Gamma \vdash M : \mathsf{M}\ A \quad \Gamma, x : A \vdash N : \mathsf{M}\ B}{\Gamma \vdash \mathsf{let}\ x{:}A \Leftarrow M\ \mathsf{in}\ N : \mathsf{M}\ B} \quad x \notin \mathrm{FV}(B)$$

Rules for natural numbers

$$\text{(nat)}\quad \vdash \mathsf{nat} : *^{v}$$

$$\text{(num)}\quad \vdash \lceil n \rceil : \mathsf{nat}$$

$$\text{(succ)}\quad \frac{\Gamma \vdash M : \mathsf{nat}}{\Gamma \vdash \mathsf{succ}\ M : \mathsf{nat}}$$

$$\text{(pred)}\quad \frac{\Gamma \vdash M : \mathsf{nat}}{\Gamma \vdash \mathsf{pred}\ M : \mathsf{nat}}$$

$$\text{(test)}\quad \frac{\Gamma \vdash M : \mathsf{nat} \quad \Gamma \vdash N_1 : \mathsf{M}\ B \quad \Gamma \vdash N_2 : \mathsf{M}\ B}{\Gamma \vdash \mathsf{if0}\ M\ N_1\ N_2 : \mathsf{M}\ B}$$

Fixpoint rule

$$\text{(fixpoint)}\quad \frac{\Gamma, x : \mathsf{M}\ A \vdash M : \mathsf{M}\ A}{\Gamma \vdash \mathsf{fix}\ x{:}\mathsf{M}\ A.\, M : \mathsf{M}\ A}$$

Fig. 2. RULES FOR MONADIC TYPE SYSTEMS (MONADS, NATURALS AND FIXPOINT RULES)

Call-by-name:

Translations that encode call-by-name evaluation in the metalanguage yield functions that map computations to computations [21, 28, 50]. The intuition is that arguments are suspended computations. Each variable access initiates evaluation of the suspended computation. There is only one rule:

$$\mathcal{R}_{cbn} = \{[*^c, *^c]\}.$$

Call-by-value:

In contrast to call-by-name, call-by-value functions take values as arguments. This is enforced by allowing only abstractions with value types in the domain position [28, 50]:

$$\mathcal{R}_{cbv} = \{[*^v, *^c]\}.$$

It is also possible to encode call-by-value in the system generated by the call-by-name specification above. The idea is to force the evaluation of the argument first, then pass the resulting value as a thunk (which corresponds to a computation, see [21, 28]).

Call-by-name and call-by-value:

This language provides a generic intermediate language for encoding both call-by-name and call-by-value programs [28] and also for strictness analyzed call-by-name programs [7, 12]. The rule set consists of the rules from both preceding cases:

$$\mathcal{R}_{mixed} = \{[*^c, *^c], [*^v, *^c]\}.$$

Full value types:

Some applications [13, 21] need to distinguish functions that are *trivial* [42] in the sense that their application causes no computational effects. With the lifting monad, such functions would be guaranteed to be *total* given any appropriately typed argument. The following specification allows for such functions by letting value types occur in the codomain position.

$$\mathcal{R}_{full} = \{[*^v, *^v], [*^c, *^v], [*^c, *^c], [*^v, *^c]\}.$$

3.2 Polymorphic systems (ML-style)

In this section, we investigate the translation of a fragment of ML into a suitable MTS. We then prove that the translation preserves typing and that ML evaluation is compatible with MTS-equality ($=_{ml}$, the reflexive, transitive, symmetric closure of $\rightarrow_{ml}$).

The starting point for a specification of an ML-style language is the specification $\mathcal{R}_{cbv}$. For concreteness, we will concentrate on a fragment of Standard ML'97 [34]. SML is a call-by-value language with impure features like references and I/O. In the fragment that we consider initially, the only effect is nontermination. Hence the monad must include the lifting monad. Later on, we add stores and require (at least) a combination of the state monad with the lifting monad. The full SML language also includes exceptions, I/O, and

types	$\tau ::= \alpha \mid \mathsf{nat} \mid \tau \to \tau$
type schemes	$\sigma ::= \tau \mid \forall\alpha.\sigma$
terms	$e ::= v \mid n$
values	$v ::= x \mid f \mid i \mid \lambda x.e \mid \mathsf{fix}\ f.e$
non-values	$n ::= e@e \mid \mathsf{let}\ x = v\ \mathsf{in}\ e \mid \mathsf{if0}\ e\ e\ e \mid \mathsf{succ}\ e \mid \mathsf{pred}\ e$

Fig. 3. A Fragment of SML'97

continuations (in the New Jersey dialect of ML). These features are not considered here, but are nevertheless tractable in the framework.

ML abstractions are modeled by the rule $[*^v, *^c]$. To express type abstractions, we add the rule $[\Box^v, *^v]$ which basically says that a type scheme is also a type. This setting ignores the fact that ML's polymorphism is predicative: in ML, type abstractions may be nested inside of type abstractions, but not inside of other type constructors.

Predicativity can be recovered in at least two ways:

(i) Add a new sort $\sharp$ and use the rules $(\Box^v, *^v, \sharp)$ and $(\Box^v, \sharp, \sharp)$ for forming type abstractions.

(ii) Add an inclusion $*^v \subseteq \Box^v$ in the style of Harper and Mitchell [25]. Now type abstractions are governed by the rule $(\Box^v, \Box^v, \Box^v)$.

Both approaches do not fit into the present framework: the specification involving $\sharp$ is not value-adhering (see Definition 26) and inclusion is not considered.[4] But still, there is no harm in admitting impredicative polymorphism in an explicitly typed intermediate language as long as type checking remains decidable.

The rule $[\Box^v, *^v]$ forces the body of a type abstraction to have a value type. Thus the MTS specification enforces in a natural way the restriction to value polymorphism present in SML'97 (this restriction ensures type soundness in the presence of side effects, see e.g., [22, 23, 49]).

In summary, here are the rules we consider.

$$\mathcal{R}_{ML} = \{[*^v, *^c], [\Box^v, *^v]\}$$

The source language for the encoding in the MTS given by $\mathcal{R}_{ML}$ is the fragment of SML'97 given in Fig. 3. A minor difference to SML'97 is the syntactic distinction between variables f bound by $\mathsf{fix}\ f.e$ and other variables x. Both are considered values, but the translation treats them differently. Furthermore, SML's $\mathsf{letrec}\ x = v\ \mathsf{in}\ e$ is syntactic sugar for $\mathsf{let}\ x \Leftarrow \mathsf{fix}\ f.v\{x := f\}\ \mathsf{in}\ e$. We have chosen the syntax with fix to simplify the translation.

The intended operational semantics is call-by-value, which we formalize by defining

[4] Actually, Pure Type Systems with Universe Inclusion have been studied and formalized in [41] and it would make sense to use them as a basis.

$$
\begin{array}{lll}
E & ::= [\,] \mid E@e \mid v@E \mid \mathsf{if0}\ E\ e\ e \mid \mathsf{succ}\ E \mid \mathsf{pred}\ E \\
(\lambda x.e)@v & \mapsto e\{x := v\} \\
(\mathsf{fix}\ f.e)@v & \mapsto e\{f := \mathsf{fix}\ f.e\}@v \\
\mathsf{let}\ x = v\ \mathsf{in}\ e & \mapsto e\{x := v\} \\
\mathsf{if0}\ 0\ e_1\ e_2 & \mapsto e_1 \\
\mathsf{if0}\ (i+1)\ e_1\ e_2 & \mapsto e_2 \\
\mathsf{succ}\ i & \mapsto (i+1) \\
\mathsf{pred}\ (i+1) & \mapsto i \\
\mathsf{pred}\ 0 & \mapsto 0 \\
l \mapsto r & \Rightarrow E[l] \to E[r]
\end{array}
$$

Fig. 4. Evaluation Relation for SML'97 Fragment

a small-step evaluation relation $\to$ using the evaluation contexts E and reduction rules shown in Fig. 4. The reduction for fix $f.e$ is slightly unusual, but it nicely captures a call-by-value fixpoint computation. Figure 5 recalls the typing rules of ML for reference. We use the notation $\tau \leq \sigma$ if $\sigma = \forall\alpha_1 \ldots \forall\alpha_n.\tau'$ and there exist $\tau_1, \ldots, \tau_n$ such that $\tau = \tau'\{\alpha_i \mapsto \tau_i\}$, and $\mathsf{gen}(\Gamma, \tau) = \forall\alpha_1 \ldots \forall\alpha_n.\tau$ where $\{\alpha_1, \ldots, \alpha_n\} = \mathrm{FV}(\tau) \setminus \mathrm{FV}(\Gamma)$ with FV denoting the set of free variables in types and type assumptions.

The translation of types and type schemes has three layers (see Fig. 6). The translation $\mathcal{C}^M$ for top-level terms generates computation types. The translation $\mathcal{V}^M$ generates value types and the translation $\mathcal{S}^M$ applies to type schemes, which also abstract over value types. This translation generates well-formed types.

Lemma 1 *Let* $G_\sigma = \alpha_1 : *^v, \ldots \alpha_n : *^v$ *where* $\{\alpha_1, \ldots, \alpha_n\} = \mathrm{FV}(\sigma)$. *Then*

(i) $G_\tau \vdash \mathcal{V}^M[\![\tau]\!] : *^v$;

(ii) $G_\tau \vdash \mathcal{C}^M[\![\tau]\!] : *^c$;

(iii) $G_\sigma \vdash \mathcal{S}^M[\![\sigma]\!] : *^v$.

The translation on terms is structured after the translation on types. It is split into the translations $\mathcal{V}^M$ of syntactic values and $\mathcal{C}^M$ of non-values. Since the translation really maps type derivations to type derivations, we assume access to a syntax-directed SML type derivation (as generated by the typing rules in Fig. 5), namely at variable accesses to $x : \forall\alpha_1 \ldots \alpha_n.\tau$ we assume access to the instantiation $\alpha_1 \mapsto \tau_1, \ldots, \alpha_n \mapsto \tau_n$, and at let expressions we assume we know the generalized type $\forall\alpha_1 \ldots \alpha_n.\tau$. We indicate the type information by superscripts and the instantiation by subscripts. Figure 7 defines the

$$\Gamma, x : \sigma \vdash_{SML} x : \tau \quad \tau \leq \sigma \qquad \Gamma, f : \tau \vdash_{SML} f : \tau \qquad \Gamma \vdash_{SML} i : \mathsf{nat}$$

$$\frac{\Gamma, x : \tau_2 \vdash_{SML} e : \tau_1}{\Gamma \vdash_{SML} \lambda x.e : \tau_2 \to \tau_1} \qquad \frac{\Gamma \vdash_{SML} e_1 : \tau_2 \to \tau_1 \quad \Gamma \vdash_{SML} e_2 : \tau_2}{\Gamma \vdash_{SML} e_1 @ e_2 : \tau_1}$$

$$\frac{\Gamma \vdash_{SML} v : \tau_1 \quad \Gamma, x : \mathsf{gen}(\Gamma, \tau_1) \vdash_{SML} e : \tau_2}{\Gamma \vdash_{SML} \mathsf{let}\ x = v\ \mathsf{in}\ e : \tau_2}$$

$$\frac{\Gamma \vdash_{SML} e_1 : \mathsf{nat} \quad \Gamma \vdash_{SML} e_2 : \tau \quad \Gamma \vdash_{SML} e_3 : \tau}{\Gamma \vdash_{SML} \mathsf{if0}\ e_1\ e_2\ e_3 : \tau}$$

$$\frac{\Gamma \vdash_{SML} e : \mathsf{nat}}{\Gamma \vdash_{SML} \mathsf{succ}\ e : \mathsf{nat}} \qquad \frac{\Gamma \vdash_{SML} e : \mathsf{nat}}{\Gamma \vdash_{SML} \mathsf{pred}\ e : \mathsf{nat}}$$

$$\frac{\Gamma, f : \tau_2 \to \tau_1 \vdash_{SML} e : \tau_2 \to \tau_1}{\Gamma \vdash_{SML} \mathsf{fix}\ f.e : \tau_2 \to \tau_1}$$

Fig. 5. Typing Rules for SML'97 Fragment

$$\begin{aligned}
\mathcal{C}^M[\![\tau]\!] &= \mathsf{M}\ \mathcal{V}^M[\![\tau]\!] \\
\mathcal{V}^M[\![\alpha]\!] &= \alpha \\
\mathcal{V}^M[\![\mathsf{nat}]\!] &= \mathsf{nat} \\
\mathcal{V}^M[\![\tau_1 \to \tau_2]\!] &= \mathcal{V}^M[\![\tau_1]\!] \to \mathcal{C}^M[\![\tau_2]\!] \\
\mathcal{S}^M[\![\tau]\!] &= \mathcal{V}^M[\![\tau]\!] \\
\mathcal{S}^M[\![\forall\alpha.\sigma]\!] &= \Pi\alpha{:}*^v.\, \mathcal{S}^M[\![\sigma]\!]
\end{aligned}$$

Fig. 6. Translation of SML'97 Types

$$
\begin{array}{ll}
\mathcal{C}^M[\![v]\!] & = \mathsf{unit}\ \mathcal{V}^M[\![v]\!] \\
\mathcal{C}^M[\![e_1^{\tau_2\to\tau_1}@e_2]\!] & = \mathsf{let}\ y_1 : \mathcal{V}^M[\![\tau_2\to\tau_1]\!] \Leftarrow \mathcal{C}^M[\![e_1]\!]\ \mathsf{in}\ \mathsf{let}\ y_2 : \mathcal{V}^M[\![\tau_2]\!] \Leftarrow \mathcal{C}^M[\![e_2]\!]\ \mathsf{in}\ y_1\ y_2 \\
\mathcal{C}^M[\![\mathsf{let}\ x^{\sigma=\forall\alpha_1\ldots\alpha_n.\tau} = v^\tau\ \mathsf{in}\ e]\!] & = (\lambda x : \mathcal{S}^M[\![\sigma]\!].\mathcal{C}^M[\![e]\!])\ (\lambda\alpha_1 : *^v \ldots \alpha_n : *^v.\mathcal{V}^M[\![v]\!]) \\
\mathcal{C}^M[\![\mathsf{if0}\ e_1\ e_2\ e_3]\!] & = \mathsf{let}\ y : \mathsf{nat} \Leftarrow \mathcal{C}^M[\![e_1]\!]\ \mathsf{in}\ \mathsf{if0}\ y\ \mathcal{C}^M[\![e_2]\!]\ \mathcal{C}^M[\![e_3]\!] \\
\mathcal{C}^M[\![\mathsf{succ}\ e]\!] & = \mathsf{let}\ y : \mathsf{int} \Leftarrow \mathcal{C}^M[\![e]\!]\ \mathsf{in}\ \mathsf{unit}\ (\mathsf{succ}\ y) \\
\mathcal{C}^M[\![\mathsf{pred}\ e]\!] & = \mathsf{let}\ y : \mathsf{int} \Leftarrow \mathcal{C}^M[\![e]\!]\ \mathsf{in}\ \mathsf{unit}\ (\mathsf{pred}\ y)
\end{array}
$$

$$
\begin{array}{ll}
\mathcal{V}^M[\![i]\!] & = \lceil i \rceil \\
\mathcal{V}^M[\![x^{\forall\alpha_1\ldots\alpha_n.\tau}_{\alpha_i\mapsto\tau_i}]\!] & = x\ \mathcal{V}^M[\![\tau_1]\!]\ \ldots\ \mathcal{V}^M[\![\tau_n]\!] \\
\mathcal{V}^M[\![f^{\tau_2\to\tau_1}]\!] & = \lambda y_2 : \mathcal{V}^M[\![\tau_2]\!].\mathsf{let}\ y_1 : \mathcal{V}^M[\![\tau_2\to\tau_1]\!] \Leftarrow f\ \mathsf{in}\ y_1\ y_2 \\
\mathcal{V}^M[\![\lambda x^\tau.e]\!] & = \lambda x : \mathcal{V}^M[\![\tau]\!].\mathcal{C}^M[\![e]\!] \\
\mathcal{V}^M[\![\mathsf{fix}\ f^{\tau_2\to\tau_1}.e]\!] & = \lambda y_2 : \mathcal{V}^M[\![\tau_2]\!].\mathsf{let}\ y_1 : \mathcal{V}^M[\![\tau_2\to\tau_1]\!] \Leftarrow \mathsf{fix}\ f : \mathcal{C}^M[\![\tau_2\to\tau_1]\!].\mathcal{C}^M[\![e]\!]\ \mathsf{in}\ y_1\ y_2
\end{array}
$$

Fig. 7. Translation of the SML'97 Fragment

translation. It generates well-formed types and is type-sound. Let

$$G_{y_1:\sigma_1,\ldots,y_n:\sigma_n} = y_1 : G_{\sigma_1}, \ldots, y_n : G_{\sigma_n}$$

where y ranges over all variables (fix-bound and other variables, $y ::= x \mid f$).

Lemma 2 *Let* $\Gamma = \{y_1 : \sigma_1, \ldots, y_n : \sigma_n\}$. *Let further* $\Gamma' = \{y_1 : A'_1, \ldots, y_n : A'_n\}$ *where* $A'_i = \mathcal{S}^M[\![A_i]\!]$ *if* y_i *is not* fix*-bound and* $A'_i = \mathcal{C}^M[\![A_i]\!]$ *otherwise. Then*

(1) $$\Gamma \vdash_{SML} e : \tau \quad \Rightarrow \quad G_\Gamma, \Gamma' \vdash \mathcal{C}^M[\![e]\!] : \mathcal{C}^M[\![\tau]\!]$$

using $\mathcal{R}_{ML}$.

Proof See Appendix C. ∎

Next, we prove that the translation is compatible with conversion.

Theorem 5 *Suppose* $e_1 \to e_2$ *in ML. Then* $\mathcal{C}^M[\![e_1]\!] =_{ml} \mathcal{C}^M[\![e_2]\!]$.

The proof and some auxiliary lemmas may be found in Appendix D.

Unfortunately, the result cannot be strengthened: it would be more satisfactory to relate to single-step MTS evaluation (see Definition 25), e.g., to have $\mathcal{C}^M[\![e_1]\!] \longmapsto_{ml} \mathcal{C}^M[\![e_2]\!]$ but that is not possible due to the translation of the let and fix constructs. In addition, the primitives succ and pred require a reduction step inside of unit . The latter could be avoided by giving the primitives a computation type.

The MTS formulation admits various extensions to the SML'97 fragment either without variation of the specification or with only slight additions.

3.2.1 References

We extend the syntax of our ML fragment by types and terms related to references and the operations thereof in the usual way.

$$\tau ::= \ldots \mid \mathsf{ref}\ \tau \mid ()$$
$$n ::= \ldots \mid \mathsf{ref}\ e \mid !\ e \mid e := e$$

That is, there are a new type constructor ref and non-value terms for creating a reference ref e, reading a reference $!\ e$, and updating the contents of a reference $e := e$. The new type constant () corresponds to the one-element type unit of ML. The typing rules are just the standard ML rules and are therefore omitted. The semantics of the operators can be defined using the monad of state transformers. However, we give a standard operational (small-step) semantics which maps a state σ and an expression to the next state and the reduced expression. Let L be a denumerable set of *locations* with $l \in L$ and let V be the set of syntactic values v.

$$\begin{array}{lll} E & ::= \ldots \mid \mathsf{ref}\ E \mid !\ E \mid E := e \mid v := E & \\ v & ::= \ldots \mid l \mid () & \textit{locations, unit value} \\ \sigma & \in L \hookrightarrow V & \end{array}$$

$$\begin{array}{lll} \sigma, E[\mathsf{ref}\ v] & \to \sigma[l \mapsto v], E[l] & l \notin \mathrm{dom}(\sigma) \\ \sigma, E[!\ l] & \to \sigma, E[\sigma(l)] & \\ \sigma, E[l := v] & \to \sigma[l \mapsto v], E[()] & \end{array}$$

The remaining reductions do not affect the store and are simple variations of the ones defined in Fig. 4.

To construct the translation we need to add three new constants to the metalanguage. These constants are the metalanguage's operators to perform the operations on references. Their types are given as follows.

$$\begin{array}{ll} \mathsf{ref}_M & : (\Pi\alpha : *^v.\alpha \to \mathsf{M}\ (\mathsf{ref}\ \alpha)) \\ !_M & : (\Pi\alpha : *^v.\mathsf{ref}\ \alpha \to \mathsf{M}\ \alpha) \\ :=_M & : (\Pi\alpha : *^v.\mathsf{ref}\ \alpha \to \alpha \to \mathsf{M}\ ()) \end{array}$$

The typing makes it clear that only values can be stored in references. The translation on types extends as follows:

$$\begin{array}{ll} \mathcal{V}^M[\![\mathsf{ref}\ \tau]\!] & = \mathsf{ref}\ \mathcal{V}^M[\![\tau]\!] \\ \mathcal{V}^M[\![()]\!] & = () \end{array}$$

The translation on terms extends correspondingly.

$$\begin{aligned}
\mathcal{C}^M[\![\mathsf{ref}\ e_{\alpha\mapsto\tau}]\!] &= \mathsf{let}\ y : \mathcal{V}^M[\![\tau]\!] \Leftarrow \mathcal{C}^M[\![e]\!]\ \mathsf{in}\ (\mathsf{ref}_M\ \mathcal{V}^M[\![\tau]\!])\ y \\
\mathcal{C}^M[\![!\ e_{\alpha\mapsto\tau}]\!] &= \mathsf{let}\ y : \mathcal{V}^M[\![\mathsf{ref}\ \tau]\!] \Leftarrow \mathcal{C}^M[\![e]\!]\ \mathsf{in}\ (!_M\ \mathcal{V}^M[\![\tau]\!])\ y \\
\mathcal{C}^M[\![e_1 := e_{2\alpha\mapsto\tau}]\!] &= \mathsf{let}\ y_1 : \mathcal{V}^M[\![\mathsf{ref}\ \tau]\!] \Leftarrow \mathcal{C}^M[\![e_1]\!]\ \mathsf{in} \\
&\qquad \mathsf{let}\ y_2 : \mathcal{C}^M[\![\tau]\!] \Leftarrow \mathcal{C}^M[\![e_2]\!]\ \mathsf{in}\ (:=_M\ \mathcal{V}^M[\![\tau]\!])\ y_1 y_2
\end{aligned}$$

The appendix A shows an example of a sound compile time transformation with references and types.

Lemma 3 *Lemma 1 extends to the new type constructors.*

Lemma 4 *Lemma 2 extends to the new constructs.*

Proof See Appendix E. ∎

3.2.2 Higher-order Polymorphism

Semi-explicit Polymorphism

Recently, there have been several proposals to extend ML with higher-order polymorphism [17,37]. The idea is that type abstractions are no longer restricted to the top level — they may appear everywhere in types — and in the type $(\Pi\alpha : *^v.\alpha) : *^v$ the variable α may range over all terms $M : *^v$ (impredicative polymorphism).

Garrigue and Remy [17] have proposed such an extension of ML with explicit type annotations, called semi-explicit polymorphism. They extend the term and type language by

$$\begin{aligned}
v &::= \ldots \mid [v : \sigma] \\
e &::= \ldots \mid \langle e \rangle \\
\tau &::= \ldots \mid [\sigma]
\end{aligned}$$

The notation $[\sigma]$ wraps a type scheme into a monotype which can instantiate type variables and which can occur nested within function types[5]. The basic idea is that $[v : \sigma]$ generalizes the inferred type of v to σ and wraps the type into a monotype so that $[v : \sigma]$ has type $[\sigma]$. The expression $\langle v \rangle$ unwraps such a canned type scheme: if $v : [\sigma]$ then $\langle v \rangle : \sigma$. Roughly, the $[v : \sigma]$ expression corresponds to type abstraction and the $\langle v \rangle$ expression corresponds to type application, but without giving the instantiation explicitly. The restriction to polymorphic values is inherited from SML'97.

The translation of this extended language into our instance of MTS is straightforward.

$$\begin{aligned}
\mathcal{V}^M[\![\,[\sigma]\,]\!] &= \mathcal{S}^M[\![\sigma]\!] \\
\mathcal{V}^M[\![\,[v^\tau : \forall\alpha_1 \ldots \alpha_n.\tau]\,]\!] &= \lambda\alpha_1 : *^v \ldots \alpha_n : *^v.\mathcal{V}^M[\![v]\!] \\
\mathcal{C}^M[\![\langle e \rangle^{[\sigma=\forall\alpha_1\ldots\alpha_n.\tau]}_{\alpha_i\mapsto\tau_i}]\!] &= \mathsf{let}\ y : \mathcal{S}^M[\![\sigma]\!] \Leftarrow \mathcal{C}^M[\![e]\!]\ \mathsf{in}\ \mathsf{unit}\ (y\ \mathcal{V}^M[\![\tau_1]\!]\ \ldots\ \mathcal{V}^M[\![\tau_n]\!])
\end{aligned}$$

[5] This glosses over the main technical point of their work, which makes type inference decidable for their system. We can safely ignore it here because we start from a given type derivation in their system.

For this translation, we can show again that typing is preserved with respect to the syntax directed system given in the type inference algorithm of Garrigue and Remy [17].

Second order polymorphic lambda calculus

In the same way, we can encode an ML-style variant of the second order polymorphic lambda calculus F_2 where SML'97 types and expressions (Fig. 3) are extended by type abstraction and type application:

$$\begin{aligned}\sigma &::= \alpha \mid \mathsf{nat} \mid \sigma \to \sigma \mid \forall\alpha.\sigma \\ v &::= \ldots \mid \Lambda\alpha.v \\ e &::= \ldots \mid \Lambda\alpha.v \mid e\{\sigma\}\end{aligned}$$

with the standard typing rules and the additional ML-style call-by-value reduction rule

$$(\Lambda\alpha.v)\{\sigma\} \mapsto v\{\alpha := \sigma\}$$

The translation of terms is extended in essentially the same way as for semi-explicit polymorphism.

$$\begin{aligned}\mathcal{V}^M[\![\Lambda\alpha.v]\!] &= \lambda\alpha : *^v.\mathcal{V}^M[\![v]\!] \\ \mathcal{C}^M[\![e^{\forall\alpha.\sigma_1}\{\sigma_2\}]\!] &= \mathsf{let}\ y : \mathcal{V}^M[\![\forall\alpha.\sigma_1]\!] \Leftarrow \mathcal{C}^M[\![e]\!]\ \mathsf{in}\ \mathsf{unit}\ (y\ \mathcal{V}^M[\![\sigma]\!])\end{aligned}$$

with the translation of value types changed to:

$$\begin{aligned}\mathcal{V}^M[\![\alpha]\!] &= \alpha \\ \mathcal{V}^M[\![\mathsf{nat}]\!] &= \mathsf{nat} \\ \mathcal{V}^M[\![\sigma_1 \to \sigma_2]\!] &= \mathcal{V}^M[\![\sigma_1]\!] \to \mathcal{C}^M[\![\sigma_2]\!] \\ \mathcal{V}^M[\![\forall\alpha.\sigma]\!] &= \Pi\alpha{:}*^v.\ \mathcal{V}^M[\![\sigma]\!]\end{aligned}$$

3.2.3 Polymorphism by name

Leroy [33] has shown that the unsoundness of the naive approach to polymorphic references in ML can be attributed to the fact that type generalization evaluates its body once and for all. The restriction to syntactic values (as enforced by our ML rule $[\Box^v, *^v]$) forces this evaluation step to be a trivial step. Another sound option, that Leroy [33] calls polymorphism by name, is to equip type generalization with a non-strict semantics and to repeat the computation whenever the corresponding type abstraction is applied. This corresponds to replacing $*^v$ by $*^c$ in the rules for type $\Box^v$ abstractions.

$$\mathcal{R}_{pbn} = \{[*^v, *^c], [*^c, *^c], [\Box^v, *^c], [\Box^v, *^v]\}$$

Let's go briefly over the uses of the four kinds of abstractions (cf. Fig. 8):

- $[*^c, *^c]$ image of a source abstraction;
- $[\Box^v, *^c]$ image of **let**: typing the innermost abstraction around the image of the header e_1;
- $[\Box^v, *^v]$ image of **let**: typing the subsequent abstractions around the image of the header e_1;

- $[*^v, *^c]$ image of let: typing the abstraction around the image of the body e_2 in case the header is polymorphic.

The modification of the translation and the proof of the preservation of typing are straightforward. Here is the translation on types:

$$\begin{aligned}
\mathcal{C}^P[\![\tau]\!] &= \mathsf{M}\ \mathcal{V}^P[\![\tau]\!] \\
\mathcal{V}^P[\![\alpha]\!] &= \alpha \\
\mathcal{V}^P[\![\mathsf{nat}]\!] &= \mathsf{nat} \\
\mathcal{V}^P[\![\tau_1 \to \tau_2]\!] &= \mathcal{C}^P[\![\tau_1]\!] \to \mathcal{C}^P[\![\tau_2]\!] \\
\mathcal{S}^P[\![\tau]\!] &= \mathcal{C}^P[\![\tau]\!] \\
\mathcal{S}^P[\![\forall\alpha.\sigma]\!] &= \Pi\alpha{:}*^v.\ \mathcal{S}^P[\![\sigma]\!]
\end{aligned}$$

For the translation of terms we need to observe the subtle change that variables (regardless whether fix-bound or not) will not be regarded as values, anymore. The encoding of call-by-value application employs the method mentioned in Sec. 3.1. The value restriction is no longer required. The revised source language is thus

$$\begin{aligned}
&\text{terms} && e ::= v \mid n \\
&\text{values} && v ::= i \mid \lambda x.e \mid \mathsf{fix}\ x.e \\
&\text{non-values} && n ::= x \mid e@e \mid \mathsf{let}\ x = e\ \mathsf{in}\ e \mid \mathsf{if0}\ e\ e\ e \mid \mathsf{succ}\ e \mid \mathsf{pred}\ e
\end{aligned}$$

Evaluation is now defined by:

$$E_{pbn} ::= [\,] \mid E_{pbn}@e \mid v@E_{pbn} \mid \mathsf{if0}\ E_{pbn}\ e\ e \mid \mathsf{succ}\ E_{pbn} \mid \mathsf{pred}\ E_{pbn}$$

$$\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 \mapsto e_2\{x := e_1\}$$

$$\ldots$$

with the remaining reductions as in the SML'97 fragment (see Fig. 4). Figure 8 defines the translation. Again, the translation preserves typing (we omit the statement of a lemma analogous to Lemma 1).

Lemma 5 *Let PBN be the syntax-directed ML type system with polymorphism by name defined by Leroy [33, Fig. 4, p.226].*

$$\Gamma \vdash_{PBN} e : \tau \quad \Rightarrow \quad G_\Gamma, \mathcal{S}^P[\![\Gamma]\!] \vdash_{\mathcal{R}_{pbn}} \mathcal{C}^P[\![e]\!] : \mathcal{C}^P[\![\tau]\!]$$

3.2.4 Call-by-name ML

As an example of a call-by-name language, we investigate a call-by-name variant of ML. In this variant, also non-functional fixpoints make sense (at least, in an extended language

$$
\begin{aligned}
\mathcal{C}^P[\![v]\!] &= \mathsf{unit}\ \mathcal{V}^P[\![v]\!] \\
\mathcal{C}^P[\![x^{\forall\alpha_1\ldots\alpha_n.\tau}_{\alpha_i\mapsto\tau_i}]\!] &= x\ \mathcal{V}^P[\![\tau_1]\!]\ \ldots\ \mathcal{V}^P[\![\tau_n]\!] \\
\mathcal{C}^P[\![e_1^{\tau_2\to\tau_1}@e_2]\!] &= \mathsf{let}\ y_1 : \mathcal{V}^P[\![\tau_2\to\tau_1]\!] \Leftarrow \mathcal{C}^P[\![e_1]\!]\ \mathsf{in}\ \mathsf{let}\ y_2 : \mathcal{V}^P[\![\tau_2]\!] \Leftarrow \mathcal{C}^P[\![e_2]\!]\ \mathsf{in}\ y_1\ (\mathsf{unit}\ y_2) \\
\mathcal{C}^P[\![\mathsf{let}\ x^{\sigma=\forall\alpha_1\ldots\alpha_n.\tau} = e_1^\tau\ \mathsf{in}\ e_2]\!] &= (\lambda x : \mathcal{S}^P[\![\sigma]\!].\mathcal{C}^P[\![e_2]\!])\ (\lambda\alpha_1 : *^v \ldots \alpha_n : *^v.\mathcal{C}^P[\![e_1]\!]) \\
\mathcal{C}^P[\![\mathsf{if0}\ e_1\ e_2\ e_3]\!] &= \mathsf{let}\ y : \mathsf{nat} \Leftarrow \mathcal{C}^P[\![e_1]\!]\ \mathsf{in}\ \mathsf{if0}\ y\ \mathcal{C}^P[\![e_2]\!]\ \mathcal{C}^P[\![e_3]\!] \\
\mathcal{C}^P[\![\mathsf{succ}\ e]\!] &= \mathsf{let}\ y : \mathsf{nat} \Leftarrow \mathcal{C}^P[\![e]\!]\ \mathsf{in}\ \mathsf{unit}\ (\mathsf{succ}\ y) \\
\mathcal{C}^P[\![\mathsf{pred}\ e]\!] &= \mathsf{let}\ y : \mathsf{nat} \Leftarrow \mathcal{C}^P[\![e]\!]\ \mathsf{in}\ \mathsf{unit}\ (\mathsf{pred}\ y) \\
\\
\mathcal{V}^P[\![i]\!] &= i \\
\mathcal{V}^P[\![\lambda x^\tau.e]\!] &= \lambda x : \mathcal{C}^P[\![\tau]\!].\mathcal{C}^P[\![e]\!] \\
\mathcal{V}^P[\![\mathsf{fix}\ x^{\tau_2\to\tau_1}.e]\!] &= \lambda y_2 : \mathcal{C}^P[\![\tau_2]\!].\mathsf{let}\ y_1 : \mathcal{V}^P[\![\tau_2\to\tau_1]\!] \Leftarrow \mathsf{fix}\ x : \mathcal{C}^P[\![\tau_2\to\tau_1]\!].\mathcal{C}^P[\![e]\!]\ \mathsf{in}\ y_1\ y_2
\end{aligned}
$$

Fig. 8. Translation for Polymorphism by Name

with sum types) so fix $x.e$ is no longer a value, but it is a redex by itself.

$$
\begin{aligned}
&\text{terms} && e ::= v \mid n \\
&\text{values} && v ::= i \mid \lambda x.e \\
&\text{non-values} && n ::= x \mid \mathsf{fix}\ x.e \mid e@e \mid \mathsf{let}\ x = e\ \mathsf{in}\ e \mid \mathsf{if0}\ e\ e\ e \mid \mathsf{succ}\ e \mid \mathsf{pred}\ e
\end{aligned}
$$

Evaluation changes in the expected way.

$$
\begin{aligned}
E_{cbn} &::= [\,] \mid E_{cbn}@e \mid \mathsf{if0}\ E_{cbn}\ e\ e \mid \mathsf{succ}\ E_{cbn} \mid \mathsf{pred}\ E_{cbn} \\
\\
(\lambda x.e_2)@e_1 &\mapsto e_2\{x := e_1\} \\
\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 &\mapsto e_2\{x := e_1\} \\
\mathsf{fix}\ x.e &\mapsto e\{x := \mathsf{fix}\ x.e\} \\
&\ldots \\
l \mapsto r &\Rightarrow E_{cbn}[l] \to_{cbn} E_{cbn}[r]
\end{aligned}
$$

It turns out that we can use the same rule set as for polymorphism by name:

$$\mathcal{R}_{cbn} = \mathcal{R}_{pbn} = \{[*^v, *^c], [*^c, *^c], [\Box^v, *^c], [\Box^v, *^v]\}$$

The translation of type schemes is identical, too.

$$
\begin{aligned}
\mathcal{C}^N[\![\tau]\!] &= \mathcal{C}^P[\![\tau]\!] \\
\mathcal{V}^N[\![\tau]\!] &= \mathcal{V}^P[\![\tau]\!] \\
\mathcal{S}^N[\![\sigma]\!] &= \mathcal{S}^P[\![\sigma]\!]
\end{aligned}
$$

$$
\begin{array}{ll}
\mathcal{C}^N[\![v]\!] & = \mathsf{unit}\ \mathcal{V}^N[\![v]\!] \\
\mathcal{C}^N[\![x^{\forall\alpha_1\ldots\alpha_n.\tau}_{\alpha_i\mapsto\tau_i}]\!] & = x\ \mathcal{V}^N[\![\tau_1]\!]\ \ldots\ \mathcal{V}^N[\![\tau_n]\!] \\
\mathcal{C}^N[\![\mathsf{fix}\ x^\tau.e]\!] & = \mathsf{fix}\ x : \mathcal{C}^N[\![\tau]\!].\mathcal{C}^N[\![e]\!] \\
\mathcal{C}^N[\![e_1^{\tau_2\to\tau_1}@e_2]\!] & = \mathsf{let}\ y_1 : \mathcal{V}^N[\![\tau_2\to\tau_1]\!] \Leftarrow \mathcal{C}^N[\![e_1]\!]\ \mathsf{in}\ y_1\ \mathcal{C}^N[\![e_2]\!] \\
\mathcal{C}^N[\![\mathsf{let}\ x^{\sigma=\forall\alpha_1\ldots\alpha_n.\tau} = e_1^\tau\ \mathsf{in}\ e_2]\!] & = (\lambda x : \mathcal{S}^N[\![\sigma]\!].\mathcal{C}^N[\![e_2]\!])\ (\lambda\alpha_1 : *^v \ldots \alpha_n : *^v.\mathcal{C}^N[\![e_1]\!]) \\
\mathcal{C}^N[\![\mathsf{if0}\ e_1\ e_2\ e_3]\!] & = \mathsf{let}\ y : \mathsf{nat} \Leftarrow \mathcal{C}^N[\![e_1]\!]\ \mathsf{in}\ \mathsf{if0}\ y\ \mathcal{C}^N[\![e_2]\!]\ \mathcal{C}^N[\![e_3]\!] \\
\mathcal{C}^N[\![\mathsf{succ}\ e]\!] & = \mathsf{let}\ y : \mathsf{nat} \Leftarrow \mathcal{C}^N[\![e]\!]\ \mathsf{in}\ \mathsf{unit}\ (\mathsf{succ}\ y) \\
\mathcal{C}^N[\![\mathsf{pred}\ e]\!] & = \mathsf{let}\ y : \mathsf{nat} \Leftarrow \mathcal{C}^N[\![e]\!]\ \mathsf{in}\ \mathsf{unit}\ (\mathsf{pred}\ y) \\
\mathcal{V}^N[\![i]\!] & = i \\
\mathcal{V}^N[\![\lambda x^\tau.e]\!] & = \lambda x : \mathcal{C}^N[\![\tau]\!].\mathcal{C}^N[\![e]\!]
\end{array}
$$

Fig. 9. Translation for call-by-name ML

The translation of terms may be found in Fig. 9. In fact, it is identical to the translation in Fig. 8, up to the cases for fix $x.e$ and $e@e$.

We can prove the following statements about the translation.

Lemma 6 *Let* $\Gamma = \{x_1 : \sigma_1, \ldots, x_n : \sigma_n\}$ *and* $\Gamma' = \{x_1 : \mathcal{S}^N[\![\sigma_1]\!], \ldots, x_n : \mathcal{S}^N[\![\sigma_n]\!]\}$. *Then*

$$\Gamma \vdash_{SML} e : \tau \quad \Rightarrow \quad G_\Gamma, \Gamma' \vdash \mathcal{C}^N[\![e]\!] : \mathcal{C}^N[\![\tau]\!] \tag{2}$$

using $\mathcal{R}_{cbn}$.

Theorem 6 *Suppose* $e_1 \to_{cbn} e_2$. *Then* $\mathcal{C}^N[\![e_1]\!] =_{ml} \mathcal{C}^N[\![e_2]\!]$.

The proofs are omitted, because they are very similar to the ones presented for the call-by-value variant. Again, we only obtain convertibility of the translated terms, which is again due to the translations of let, fix, and the primitives.

3.3 Encodings of F_ω

Finally, we give an outline of call-by-name and call-by-value translations for F_ω [19], also known as the higher-order lambda calculus. Figure 10 defines the syntax of the source language. We omit the standard formation rules for constructor contexts Δ, term contexts Θ, constructors $\Delta \triangleright \sigma : \kappa$, constructor conversion $\Delta \triangleright \sigma_1 = \sigma_2 : \kappa$, and terms $\Delta; \Theta \triangleright e : \sigma$.

3.3.1 Standard Call-by-Name Encoding

Figure 11 shows a call-by-name encoding of F_ω. In order to prove that this translation preserves typing, we need the following specification:

$$\mathcal{R}_{Fcbn} = \{(\Box^v, \Box^v, \Box^v), [*^c, *^c], [\Box^v, *^c]\}$$

$$
\begin{array}{lll}
\text{kinds} & \kappa ::= \Omega \mid \kappa \Rightarrow \kappa \\
\text{constructors} & \sigma ::= \alpha \mid \sigma \to \sigma \mid \forall \alpha : \kappa.\sigma \mid \lambda \alpha : \kappa.\sigma \mid \sigma\{\sigma\} \\
\text{terms} & e ::= x \mid \lambda x : \sigma.e \mid e @ e \mid \Lambda \alpha : \kappa.e \mid e\{\sigma\}
\end{array}
$$

Fig. 10. Syntax of F_ω

Translation of kinds

$$
\begin{aligned}
\mathcal{K}[\![\Omega]\!] &= *^v \\
\mathcal{K}[\![\kappa_1 \Rightarrow \kappa_2]\!] &= \mathcal{K}[\![\kappa_1]\!] \to \mathcal{K}[\![\kappa_2]\!]
\end{aligned}
$$

Call-by-name translation of constructors

$$
\begin{array}{lll}
\mathcal{T}[\![\alpha^\Omega]\!] = \mathsf{M}\ \alpha & & \\
\mathcal{T}[\![\alpha^{\kappa_1 \Rightarrow \kappa_2}]\!] = \alpha & & \mathcal{V}[\![\alpha]\!] = \alpha \\
\mathcal{T}[\![\sigma_1 \to \sigma_2]\!] = \mathsf{M}\ (\mathcal{T}[\![\sigma_1]\!] \to \mathcal{T}[\![\sigma_2]\!]) & & \mathcal{V}[\![\sigma_1 \to \sigma_2]\!] = \mathcal{T}[\![\sigma_1]\!] \to \mathcal{T}[\![\sigma_2]\!] \\
\mathcal{T}[\![\forall \alpha : \kappa.\sigma]\!] = \mathsf{M}\ (\Pi \alpha{:}\mathcal{K}[\![\kappa]\!].\ \mathcal{T}[\![\sigma]\!]) & & \mathcal{V}[\![\forall \alpha : \kappa.\sigma]\!] = \Pi \alpha{:}\mathcal{K}[\![\kappa]\!].\ \mathcal{T}[\![\sigma]\!] \\
\mathcal{T}[\![\lambda \alpha : \kappa.\sigma]\!] = \lambda \alpha : \mathcal{K}[\![\kappa]\!].\mathcal{T}[\![\sigma]\!] & & \mathcal{V}[\![\lambda \alpha : \kappa.\sigma]\!] = \lambda \alpha : \mathcal{K}[\![\kappa]\!].\mathcal{V}[\![\sigma]\!] \\
\mathcal{T}[\![\sigma_1\{\sigma_2\}]\!] = \mathcal{T}[\![\sigma_1]\!]\ (\mathcal{V}[\![\sigma_2]\!]) & & \mathcal{V}[\![\sigma_1\{\sigma_2\}]\!] = \mathcal{V}[\![\sigma_1]\!]\ (\mathcal{V}[\![\sigma_2]\!])
\end{array}
$$

Call-by-name translation of terms

$$
\begin{aligned}
\mathcal{O}[\![x]\!] &= x \\
\mathcal{O}[\![\lambda x : \sigma.e]\!] &= \mathsf{unit}\ (\lambda x : \mathcal{T}[\![\sigma]\!].\mathcal{O}[\![e]\!]) \\
\mathcal{O}[\![e_1^{\sigma_2 \to \sigma_1} @ e_2]\!] &= \mathsf{let}\ y : \mathcal{V}[\![\sigma_2 \to \sigma_1]\!] \Leftarrow \mathcal{O}[\![e_1]\!]\ \mathsf{in}\ y\ (\mathcal{O}[\![e_2]\!]) \\
\mathcal{O}[\![\Lambda \alpha : \kappa.e]\!] &= \mathsf{unit}\ (\lambda \alpha : \mathcal{K}[\![\kappa]\!].\mathcal{O}[\![e]\!]) \\
\mathcal{O}[\![e^{\forall \alpha:\kappa.\sigma_1}\{\sigma_2\}]\!] &= \mathsf{let}\ y : \mathcal{V}[\![\forall \alpha : \kappa.\sigma_1]\!] \Leftarrow \mathcal{O}[\![e]\!]\ \mathsf{in}\ y\ (\mathcal{V}[\![\sigma]\!])
\end{aligned}
$$

Fig. 11. Standard call-by-name encoding of F_ω

Objects

$$O ::= x \mid \mathsf{unit}\ W \mid \mathsf{let}\ x : \hat{W} \Leftarrow O\ \mathsf{in}\ O \mid x\ O \mid x\ \hat{W} \mid W\ O \mid W\ \hat{W}$$

Values

$$W ::= \lambda x : \hat{T}.O \mid \lambda\alpha : \hat{K}.O$$

Value constructors

$$\hat{W} ::= \alpha \mid \hat{T} \to \hat{T} \mid \Pi\alpha{:}\hat{K}.\,\hat{T} \mid \lambda\alpha : \hat{K}.\hat{W} \mid \hat{W}\ \hat{W}$$

Computation constructors

$$\hat{T} ::= \alpha^{\hat{K}_1 \to \hat{K}_2} \mid \mathsf{M}\ \hat{W} \mid \lambda\alpha : \hat{K}.\hat{T} \mid \hat{T}\ \hat{W}$$

Kinds

$$\hat{K} ::= *^v \mid \hat{K} \to \hat{K}$$

Fig. 12. Image of the standard call-by-name translation

Lemma 7 *For a constructor context* $\Delta = \alpha_1 : \kappa_1, \ldots, \alpha_n : \kappa_n$ *let* $G_\Delta = \alpha_1 : \mathcal{K}[\![\kappa_1]\!], \ldots, \alpha_n : \mathcal{K}[\![\kappa_n]\!]$. *For a term context* $\Theta = x_1 : \sigma_1, \ldots, x_n : \sigma_n$ *let* $G_\Theta = x_1 : \mathcal{T}[\![\sigma_1]\!], \ldots, x_n : \mathcal{T}[\![\sigma_n]\!]$.

(i) If $\Delta \triangleright \sigma : \kappa$ *then* $G_\Delta \vdash \mathcal{V}[\![\sigma]\!] : \mathcal{K}[\![\kappa]\!]$.

(ii) If $\Delta; \Theta \triangleright e : \sigma$ *then* $G_\Delta, G_\Theta \vdash \mathcal{O}[\![e]\!] : \mathcal{T}[\![\sigma]\!]$.

Harper and Lillibridge [22, 24] have defined a standard call-by-name semantics for a superset of the calculus considered here. For their small-step operational semantics $\to_{std-cbn}$ (see Appendix F.1), we can show that the translation is compatible with single-step reduction for MTS (see Definition 25).

Theorem 7 *Suppose* $\Delta; \Theta \triangleright e_1 : \sigma_1$ *and* $e_1 \to_{std-cbn} e_2$ *then* $\mathcal{O}[\![e_1]\!] \longmapsto_{ml} \mathcal{O}[\![e_2]\!]$.

In order to characterize the connection between the source calculus and the image precisely, we work towards the definition of an inverse translation. The first step is the characterization of the image of the translation.

Lemma 8 *The grammar in Figure 12 characterizes the closure of the image of the standard call-by-name encoding under* $=_{ml}$.

Figure 13 defines the inverse translation.

Lemma 9 *Suppose* $\Gamma \vdash O : \hat{T}$. *Then there exists a constructor context* Δ *and a term context* Θ *such that* $\Delta; \Theta \triangleright \mathcal{O}^{-1}[\![O]\!] : \mathcal{T}^{-1}[\![\hat{T}]\!]$.

Theorem 8 *Suppose* $\Gamma \vdash O_1 : \hat{T}$ *and* $O_1 \longmapsto_{ml} O_2$ *then* $\mathcal{O}^{-1}[\![O_1]\!] =_\beta \mathcal{O}^{-1}[\![O_2]\!]$.

The reduction rule μ_3 is the culprit for having $=_\beta$ instead of $=_{std-cbn}$ or even $\to_{std-cbn}$.

Objects

$$\mathcal{O}^{-1}[\![x : \hat{T} : *^c]\!] = x$$
$$\mathcal{O}^{-1}[\![\mathsf{unit}\ W]\!] = \mathcal{W}^{-1}[\![W]\!]$$
$$\mathcal{O}^{-1}[\![\mathsf{let}\ x : \hat{W} \Leftarrow O_1\ \mathsf{in}\ O_2]\!] = (\lambda x : \mathcal{V}^{-1}[\![\hat{W}]\!].\mathcal{O}^{-1}[\![O_2]\!])@\mathcal{O}^{-1}[\![O_1]\!]$$
$$\mathcal{O}^{-1}[\![x\ O]\!] = x@\mathcal{O}^{-1}[\![O]\!]$$
$$\mathcal{O}^{-1}[\![x\ \hat{W}]\!] = x@\mathcal{V}^{-1}[\![\hat{W}]\!]$$
$$\mathcal{O}^{-1}[\![W\ O]\!] = \mathcal{W}^{-1}[\![W]\!]@\mathcal{O}^{-1}[\![O]\!]$$
$$\mathcal{O}^{-1}[\![W\ \hat{W}]\!] = \mathcal{W}^{-1}[\![W]\!]@\mathcal{V}^{-1}[\![\hat{W}]\!]$$

Values

$$\mathcal{W}^{-1}[\![\lambda x : \hat{T}.O]\!] = \lambda x : \mathcal{T}^{-1}[\![\hat{T}]\!].\mathcal{O}^{-1}[\![O]\!]$$
$$\mathcal{W}^{-1}[\![\lambda \alpha : \hat{K}.O]\!] = \lambda \alpha : \mathcal{K}^{-1}[\![\hat{K}]\!].\mathcal{O}^{-1}[\![O]\!]$$

Value constructors

$$\mathcal{V}^{-1}[\![\alpha]\!] = \alpha$$
$$\mathcal{V}^{-1}[\![\hat{T}_1 \to \hat{T}_2]\!] = \mathcal{T}^{-1}[\![\hat{T}_1]\!] \to \mathcal{T}^{-1}[\![\hat{T}_2]\!]$$
$$\mathcal{V}^{-1}[\![\Pi \alpha{:}\hat{K}.\,\hat{T}]\!] = \forall \alpha : \mathcal{K}^{-1}[\![\hat{K}]\!].\mathcal{T}^{-1}[\![\hat{T}]\!]$$
$$\mathcal{V}^{-1}[\![\lambda \alpha : \hat{K}.\hat{W}]\!] = \lambda \alpha : \mathcal{K}^{-1}[\![\hat{K}]\!].\mathcal{V}^{-1}[\![\hat{W}]\!]$$
$$\mathcal{V}^{-1}[\![\hat{W}_1\ \hat{W}_2]\!] = \mathcal{V}^{-1}[\![\hat{W}_1]\!]@\mathcal{V}^{-1}[\![\hat{W}_2]\!]$$

Computation constructors

$$\mathcal{T}^{-1}[\![\alpha]\!] = \alpha$$
$$\mathcal{T}^{-1}[\![\mathsf{M}\ \hat{W}]\!] = \mathcal{V}^{-1}[\![\hat{W}]\!]$$
$$\mathcal{T}^{-1}[\![\lambda \alpha : \hat{K}.\hat{T}]\!] = \lambda \alpha : \mathcal{K}^{-1}[\![\hat{K}]\!].\mathcal{T}^{-1}[\![\hat{T}]\!]$$
$$\mathcal{T}^{-1}[\![\hat{T}\ \hat{W}]\!] = \mathcal{T}^{-1}[\![\hat{T}]\!]@\mathcal{V}^{-1}[\![\hat{W}]\!]$$

Kinds

$$\mathcal{K}^{-1}[\![*^v]\!] = \Omega$$
$$\mathcal{K}^{-1}[\![\hat{K}_1 \to \hat{K}_2]\!] = \mathcal{K}^{-1}[\![\hat{K}_1]\!] \Rightarrow \mathcal{K}^{-1}[\![\hat{K}_2]\!]$$

Fig. 13. Inverse of the standard call-by-name translation

Call-by-value translation of constructors

$$\mathcal{T}^V[\![\sigma_1 \to \sigma_2]\!] = \mathsf{M}\ (\mathcal{V}^V[\![\sigma_1]\!] \to \mathcal{T}^V[\![\sigma_2]\!]) \qquad \mathcal{V}^V[\![\sigma_1 \to \sigma_2]\!] = \mathcal{V}^V[\![\sigma_1]\!] \to \mathcal{T}^V[\![\sigma_2]\!]$$

Call-by-value translation of terms

$$\begin{aligned}
\mathcal{O}^V[\![x]\!] &= \mathsf{unit}\ x \\
\mathcal{O}^V[\![\lambda x : \sigma.e]\!] &= \mathsf{unit}\ (\lambda x : \mathcal{V}^V[\![\sigma]\!].\mathcal{O}^V[\![e]\!]) \\
\mathcal{O}^V[\![e_1^{\sigma_2 \to \sigma_1} @ e_2]\!] &= \mathsf{let}\ y_1 : \mathcal{V}^V[\![\sigma_2 \to \sigma_1]\!] \Leftarrow \mathcal{O}^V[\![e_1]\!]\ \mathsf{in}\ \mathsf{let}\ y_2 : \sigma_2 \Leftarrow \mathcal{O}^V[\![e_2]\!]\ \mathsf{in}\ y_1\ y_2
\end{aligned}$$

Fig. 14. Standard call-by-value encoding of F_ω

3.3.2 Standard Call-by-Value Encoding

Next, we consider a call-by-value translation. The changes with respect to the call-by-name version are minimal: the translation of the function constructor changes because call-by-value functions take values as arguments. This causes a reclassification of x as a value and changes in the translations of (value) abstraction and application. Figure 14 gives those parts that have changed with respect to Fig. 11. The MTS specification for the image of the call-by-value translation changes only the rule for value abstractions (with respect to the call-by-name specification $\mathcal{R}_{Fcbn}$).

$$\mathcal{R}_{Fcbv} = \{(\Box^v, \Box^v, \Box^v), [*^v, *^c], [\Box^v, *^c]\}$$

Again, we can prove our two standard results.

Lemma 10 *For a constructor context* $\Delta = \alpha_1 : \kappa_1, \ldots, \alpha_n : \kappa_n$ *let* $G_\Delta = \alpha_1 : \mathcal{K}[\![\kappa_1]\!], \ldots, \alpha_n : \mathcal{K}[\![\kappa_n]\!]$. *For a term context* $\Theta = x_1 : \sigma_1, \ldots, x_n : \sigma_n$ *let* $G_\Theta = x_1 : \mathcal{V}^V[\![\sigma_1]\!], \ldots, x_n : \mathcal{V}^V[\![\sigma_n]\!]$.

(i) If $\Delta \triangleright \sigma : \kappa$ *then* $G_\Delta \vdash \mathcal{V}^V[\![\sigma]\!] : \mathcal{K}[\![\kappa]\!]$.

(ii) If $\Delta; \Theta \triangleright e : \sigma$ *then* $G_\Delta, G_\Theta \triangleright \mathcal{O}^V[\![e]\!] : \mathcal{T}^V[\![\sigma]\!]$.

Theorem 9 *Suppose* $\Delta; \Theta \triangleright e_1 : \sigma_1$ *and* $e_1 \to_{std-cbv} e_2$ *then* $\mathcal{O}^V[\![e_1]\!] \longmapsto_{ml} \mathcal{O}^V[\![e_2]\!]$.

See Appendix F.2 for the definition of $\to_{std-cbv}$.

3.3.3 ML-style Call-by-Value Encoding

It is also possible to consider an ML-style variant of the call-by-value semantics. The difference to the standard call-by-value semantics lies in the fact that ML performs computations under type abstraction. The ML-style translation presents only a few changes with respect to the standard call-by-value encoding in Fig. 14. Therefore, Figure 15 presents only the changes, which amount to the cases involving the constructor $\forall\alpha : \kappa.\sigma$.

We can prove our two standard results.

Lemma 11 *For a constructor context* $\Delta = \alpha_1 : \kappa_1, \ldots, \alpha_n : \kappa_n$ *let* $G_\Delta = \alpha_1 : \mathcal{K}[\![\kappa_1]\!], \ldots, \alpha_n : \mathcal{K}[\![\kappa_n]\!]$. *For a term context* $\Theta = x_1 : \sigma_1, \ldots, x_n : \sigma_n$ *let* $G_\Theta = x_1 : \mathcal{V}^{MV}[\![\sigma_1]\!], \ldots, x_n : \mathcal{V}^{MV}[\![\sigma_n]\!]$.

ML-style call-by-value translation of constructors

$$\mathcal{T}^{MV}[\![\forall\alpha:\kappa.\sigma]\!] = \Pi\alpha{:}\mathcal{K}[\![\kappa]\!].\,\mathcal{T}^{MV}[\![\sigma]\!] \qquad \mathcal{V}^{MV}[\![\forall\alpha:\kappa.\sigma]\!] = \Pi\alpha{:}\mathcal{K}[\![\kappa]\!].\,\mathcal{T}^{MV}[\![\sigma]\!]$$

ML-style call-by-value translation of terms

$$\mathcal{O}^{MV}[\![\Lambda\alpha:\kappa.e]\!] = \lambda\alpha:\mathcal{K}[\![\kappa]\!].\mathcal{O}^{MV}[\![e]\!]$$
$$\mathcal{O}^{MV}[\![e^{\forall\alpha:\kappa.\sigma_1}\{\sigma_2\}]\!] = \mathcal{O}^{MV}[\![e]\!]\ (\mathcal{T}^{MV}[\![\sigma]\!])$$

Fig. 15. ML-style call-by-value encoding of F_ω

(i) If $\Delta \triangleright \sigma : \kappa$ *then* $G_\Delta \vdash \mathcal{V}^{MV}[\![\sigma]\!] : \mathcal{K}[\![\kappa]\!]$.
(ii) If $\Delta;\Theta \triangleright e : \sigma$ *then* $G_\Delta, G_\Theta \triangleright \mathcal{O}^{MV}[\![e]\!] : \mathcal{T}^{MV}[\![\sigma]\!]$.

Theorem 10 *Suppose* $\Delta;\Theta \triangleright e_1 : \sigma_1$ *and* $e_1 \rightarrow_{ml-cbv} e_2$ *then* $\mathcal{O}^{MV}[\![e_1]\!] =_{ml} \mathcal{O}^{MV}[\![e_2]\!]$.

See Appendix F.3 for the definition of $\rightarrow_{ml-cbv}$.

We can strengthen the result if we restrict the source language to only admit value polymorphism as in

$$e ::= \ldots \mid \Lambda\alpha:\kappa.v$$

In this case, the reductions are compatible, again.

Theorem 11 *Suppose* e_1 *adheres to the value restriction,* $\Delta;\Theta \triangleright e_1 : \sigma_1$, *and* $e_1 \rightarrow_{ml-cbv} e_2$ *then* $\mathcal{O}^{MV}[\![e_1]\!] \longmapsto_{ml} \mathcal{O}^{MV}[\![e_2]\!]$.

3.3.4 ML-style Call-by-Name Encoding

It is possible to define a call-by-name variant of the ML-style call-by-value encoding considered above. Harper and Lillibridge [24] show that this semantics actually coincides with the standard call-by-name semantics on closed terms of base type. Therefore, we do not consider it here in any depth.

3.3.5 FLINT

Shao [45] has considered an extension of F_ω with the ML-style call-by-value semantics as an intermediate language for compilation. The main differences to our system (Sec. 3.3.3) are the following.

(i) The term language of FLINT is restricted to A-normal form. In A-normal form, the results of all non-trivial computatations have to be named [16]. Therefore, FLINT restricts function application to the form $\mathsf{let}\ x = v@v\ \mathsf{in}\ e$ (this construct does not generate polymorphism) and relies on a preceding transformation to A-normal form. This restriction slightly simplifies the translation of applications:

$$\mathcal{O}^{MV}[\![\mathsf{let}\ x = v_1^{\sigma_2\rightarrow\sigma_1}@v_2\ \mathsf{in}\ e]\!] = \mathsf{let}\ x : \mathcal{V}^{MV}[\![\sigma_1]\!] \Leftarrow \mathcal{O}^{\prime MV}[\![v_1]\!]\ (\mathcal{O}^{\prime MV}[\![v_2]\!])\ \mathsf{in}\ \mathcal{O}^{MV}[\![e]\!]$$

where $\mathcal{O}^{\prime MV}[\![\cdot]\!]$ drops the leading $\mathsf{unit}\ \cdot$ from $\mathcal{O}^{MV}[\![\cdot]\!]$ restricted to values.

(ii) FLINT has several built-in primitives. These can be dealt with just as demonstrated for succ and pred in the ML encoding (see Sec. 3.2).

(iii) FLINT's type system has predicative polymorphism. It achieves this by distinguishing between constructors and types and by having an explicit type forming operator T that maps a constructor of kind Ω to a type (as introduced by Harper and Morrisett [26]).

One way to model this would be to extend the set of sorts by a sort $\diamond$ with $\diamond : \Box^v$ for the constructors and add a rule like

$$\frac{\Gamma \vdash A : \diamond}{\Gamma \vdash T(A) : *^v}$$

to the MTS framework. This way, the specification would be

$$\{[*^v, *^c], [\Box^v, *^c], (\diamond, \diamond, \diamond)\}$$

which is well-behaved in the current framework.

(iv) There is a new kind of sequence kinds $\kappa \otimes \kappa$ in FLINT (to express the type of a module) which is not present in MTS, but could be added without much difficulty.

4 Properties of Monadic Type Systems

The meta-theory of Monadic Type Systems is complicated by two major hurdles, namely the non-termination of ϕ-reduction and the non-confluence of μ-reduction. Nevertheless, MTSs enjoy some important properties such as Subject Reduction and Classification for injective specifications. Besides, a large class of MTSs is strongly normalizing w.r.t. $\beta\iota\mu$-reduction and has decidable type-checking.

4.1 Basic properties

Basic properties such as the Substitution Lemma or the Generation Lemma, which do not rely on confluence nor normalization, are proved in exactly the same way as for Pure Type Systems.

The first Lemma states that judgements are closed under substitution—substitution is extended to pseudo-contexts in the obvious way.

Lemma 12 (Substitution) *Assume* $\Gamma, x : A, \Delta \vdash B : C$ *and* $\Gamma \vdash a : A$. *Then also* $\Gamma, \Delta\{x := a\} \vdash B\{x := a\} : C\{x := a\}$.

Proof By induction on the derivation of $\Gamma, x : A, \Delta \vdash B : C$. ∎

The second Lemma states that introducing new assumptions do not affect derivability—we write $\Delta \supseteq \Gamma$ if $(x : A) \in \Delta$ whenever $(x : A) \in \Gamma$.

Lemma 13 (Thinning) *If* $\Gamma \vdash A : B$ *and* $\Delta \supseteq \Gamma$ *is legal then* $\Delta \vdash A : B$.

Proof By induction on the derivation of $\Gamma \vdash A : B$. ∎

Conversely, one can remove unused assumptions without affecting derivability. However, this result is surprinsingly difficult to prove and is postponed until the end of this Section.

$$
\begin{array}{lll}
\Gamma \vdash s : C & \Rightarrow & \exists (s, s') \in \mathcal{A}.\ C =_{ml} s' \\
\Gamma \vdash x : C & \Rightarrow & \exists s \in \mathcal{S}, D \in \mathcal{T}.\ C =_{ml} D \quad \wedge \quad (x : D) \in \Gamma \\
 & & \quad \wedge \quad \Gamma \vdash D : s \quad \wedge \quad x \in \mathcal{V}^s \\
\Gamma \vdash \lambda x{:}A.\, b : C & \Rightarrow & \exists s \in \mathcal{S}, B \in \mathcal{T}.\ C =_{ml} \Pi x{:}A.\, B \quad \wedge \quad \Gamma, x : A \vdash b : B \\
 & & \quad \wedge \quad \Gamma \vdash \Pi x{:}A.\, B : s \\
\Gamma \vdash \Pi x{:}A.\, B : C & \Rightarrow & \exists (s_1, s_2, s_3) \in \mathcal{R}.\ C =_{ml} s_3 \quad \wedge \quad \Gamma \vdash A : s_1 \\
 & & \quad \wedge \quad \Gamma, x : A \vdash B : s_2 \\
\Gamma \vdash F\, a : C & \Rightarrow & \exists x \in \mathcal{V}, A, B \in \mathcal{T}.\ C =_{ml} B\{x := a\} \\
 & & \quad \wedge \quad \Gamma \vdash F : \Pi x{:}A.\, B \quad \wedge \quad \Gamma \vdash a : A \\
\Gamma \vdash \mathsf{fix}\ x{:}A.\, b : C & \Rightarrow & \exists B \in \mathcal{T}. C =_{ml} A \quad \wedge \quad A \equiv \mathsf{M}\ B \quad \wedge \quad \Gamma, x : A \vdash b : A \\
\Gamma \vdash \mathsf{let}\ x{:}A \Leftarrow M\ \mathsf{in}\ N : C & \Rightarrow & \exists B \in \mathcal{T}.\ C =_{ml} \mathsf{M}\ B \quad \wedge \quad \Gamma, x : A \vdash N : \mathsf{M}\ B \\
 & & \quad \wedge \quad \Gamma \vdash M : \mathsf{M}\ A \\
\Gamma \vdash \mathsf{unit}\ M : C & \Rightarrow & \exists B \in \mathcal{T}.\ C =_{ml} \mathsf{M}\ B \quad \wedge \quad \Gamma \vdash M : B \\
 & & \quad \wedge \quad \Gamma \vdash \mathsf{M}\ B : *^c \\
\Gamma \vdash \mathsf{M}\ A : C & \Rightarrow & C =_{ml} *^c \quad \wedge \quad \exists s \in \{*^c, *^v\}.\ \Gamma \vdash A : s \\
\Gamma \vdash \mathsf{nat} : C & \Rightarrow & C =_{ml} *^v \\
\Gamma \vdash \lceil n \rceil : C & \Rightarrow & C =_{ml} \mathsf{nat} \\
\Gamma \vdash \mathsf{succ}\ M : C & \Rightarrow & C =_{ml} \mathsf{nat} \quad \wedge \quad \Gamma \vdash M : \mathsf{nat} \\
\Gamma \vdash \mathsf{pred}\ M : C & \Rightarrow & C =_{ml} \mathsf{nat} \quad \wedge \quad \Gamma \vdash M : \mathsf{nat} \\
\Gamma \vdash \mathsf{if0}\ M\ M_1\ M_2 : C & \Rightarrow & \exists B \in \mathcal{T}.\ C =_{ml} \mathsf{M}\ B \quad \wedge \quad \Gamma \vdash M : \mathsf{nat} \\
 & & \quad \wedge \quad \Gamma \vdash M_1 : B \quad \wedge \quad \Gamma \vdash M_2 : B
\end{array}
$$

Fig. 16. Generation

The next Lemma provides an analysis of the possible derivations of a judgement. The lemma is used extensively throughout the paper.

Lemma 14 (Generation) *See Figure 16.*

Proof By induction on the structure of derivations using the Thinning Lemma. ■

The last result states that types are correct, in the sense that if $A : B$ then either B is a sort or $B : s$ for some sort s.

Lemma 15 (Correctness of types) *If* $\Gamma \vdash A : B$ *then either* $B \in \mathcal{S}$ *or* $\exists s \in \mathcal{S}.\ \Gamma \vdash B : s$.

Proof By induction on $\Gamma \vdash A : B$. ∎

4.2 Strong normalization

Fixpoint reduction, i.e. ϕ-reduction is not normalizing hence we cannot expect a MTS to be *ml*-strongly normalizing. However MTSs may still be $\beta\iota\mu$-strongly normalizing. The purpose of this Subsection is to define such a class of MTSs. The actual definition of the class is determined by the technique used to prove strong normalisation rather than by any intrinsic criterion. If we view MTS-specifications as a subclass of PTS-specifications, this class corresponds to those specifications which may be embedded in Barendregt's λ-cube [3, 4].

Definition 12 Let $\mathbf{S} = (\mathcal{S}, *^v, *^c, \mathcal{A}, \mathcal{R})$ be a MTS-specification.

(i) The set $\mathcal{S}^\top$ of *top-sorts* is defined as $\{s \in \mathcal{S} \mid \forall s' \in \mathcal{S}.\ (s, s') \notin \mathcal{A}\}$.

(ii) The set $\mathcal{S}^\bot$ of *bottom-sorts* is defined as $\{s \in \mathcal{S} \mid \forall s' \in \mathcal{S}.\ (s,' s) \notin \mathcal{A}\}$.

(iii) $\mathbf{S}$ is *cubic* if
   (a) $\mathcal{S} = \mathcal{S}^\top \cup \mathcal{S}^\bot$,
   (b) $*^v, *^c \in \mathcal{S}^\bot$,
   (c) for every $(s_1, s_2, s_3) \in \mathcal{R}$, $s_2 \in \mathcal{S}^\bot \Leftrightarrow s_3 \in \mathcal{S}^\bot$.

Our method to prove strong normalization is to define a reduction-preserving translation from cubic MTSs to the Calculus of Constructions with Fixpoints λC_{fix} [1, 5]. The translation could be generalized to specifications that do not comply with requirement (a) of the above definition; in that case the target language would be a Logical Pure Type System with Fixpoints at the $*$ level. However, we would need to prove such systems to be strongly normalizing in order to conclude—a method is suggested in [5] but is still in need to be carried out in detail. Following standard practice, we write $\lambda\mathbf{S} \models \mathrm{SN}(\beta\iota\mu)$ iff every legal term in $\lambda\mathbf{S}$ is $\beta\iota\mu$-strongly normalizing.

Theorem 13 *If* $\mathbf{S}$ *is a cubic specification, then* $\lambda\mathbf{S} \models \mathrm{SN}(\beta\iota\mu)$.

Proof For the sake of simplicity, we consider a slightly modified syntax where if0 expressions are labelled by their result types, i.e. are of the form $\mathsf{if0}_A\ M\ M_1\ M_2$. This enables us to encode if0 impredicatively. For the same reason, we only consider the rule

$\mathsf{pred}\ (\mathsf{succ}\ M) \to_\iota M$ for M a numeral. The translation $\lceil . \rceil$ defined inductively as follows:

$$\begin{aligned}
\lceil x \rceil &= x \\
\lceil s \rceil &= s \\
\lceil \lambda x : A.t \rceil &= \lambda x : \lceil A \rceil . \lceil t \rceil \\
\lceil \Pi x : A.B \rceil &= \Pi x : \lceil A \rceil . \lceil B \rceil \\
\lceil t\ u \rceil &= \lceil t \rceil\ \lceil u \rceil \\
\lceil \mathsf{let}\ x{:}A \Leftarrow N\ \mathsf{in}\ M \rceil &= (\lambda x{:}\lceil A \rceil .\ \lceil M \rceil)\ \lceil N \rceil \\
\lceil \mathsf{unit}\ t \rceil &= \lceil t \rceil \\
\lceil \mathsf{M}\ A \rceil &= \lceil A \rceil \\
\lceil \mathsf{fix}\ x{:}A.\ M \rceil &= \mathsf{fix}\ x{:}\lceil A \rceil .\ \lceil M \rceil \\
\lceil \mathsf{nat} \rceil &= \Pi\alpha{:}*.\ \alpha \to (\alpha \to \alpha) \to \alpha \\
\lceil \lceil n \rceil \rceil &= \underline{n} \\
\lceil \mathsf{succ}\ M \rceil &= \lambda\alpha{:}*.\ \lambda x{:}\alpha.\ \lambda f{:}\alpha \to \alpha.\ f\ (\lceil M \rceil\ \alpha\ x\ f) \\
\lceil \mathsf{pred}\ M \rceil &= \underline{\mathsf{pred}}\ \lceil M \rceil \\
\lceil \mathsf{if0}_A\ M\ M_1\ M_2 \rceil &= \lceil M \rceil\ \lceil A \rceil\ \lceil M_1 \rceil\ (\lambda _ {:} \lceil A \rceil .\ \lceil M_2 \rceil)
\end{aligned}$$

where $\underline{n}$ and $\underline{\mathsf{pred}}$ are the impredicative encodings of n and predecessor respectively. [20, page 90]. The translation $\lceil . \rceil$ preserves conversion and maps legal terms of $\lambda\mathbf{S}$ to legal terms of λC_{fix}. In addition $\lceil . \rceil$ maps infinite $\beta\mu\iota$-reduction sequences with infinitely many $\beta\mu$-steps to infinite $\beta\beta'$-reduction sequences, where β'-reduction is defined by the contraction rule:

$$P\ ((\lambda x{:}A.\ Q)\ R) \quad \to_{\beta'} \quad (\lambda x{:}A.\ P\ Q)\ R \quad \text{provided } x \notin \mathrm{FV}(P)$$

Conclude by proving that $\mathrm{SN}(\beta\beta') = \mathrm{SN}(\beta)$, see Appendix, and by noting that an infinite $\beta\mu\iota$-reduction sequence must contain infinitely many $\beta\mu$-steps. ∎

4.3 Confluence

μ-reduction is not confluent on pseudo-terms, as illustrated by the following example: [6]

$$\begin{aligned}
\mathsf{let}\ x{:}A \Leftarrow M\ \mathsf{in}\ x &=_\mu \mathsf{let}\ x{:}A \Leftarrow (\mathsf{let}\ y{:}B \Leftarrow M\ \mathsf{in}\ (\mathsf{unit}\ y))\ \mathsf{in}\ x \\
&=_\mu \mathsf{let}\ y{:}B \Leftarrow M\ \mathsf{in}\ (\mathsf{let}\ x{:}A \Leftarrow (\mathsf{unit}\ y)\ \mathsf{in}\ x) \\
&=_\mu \mathsf{let}\ x{:}B \Leftarrow M\ \mathsf{in}\ x
\end{aligned}$$

[6] However, we believe that the failure of confluence for monadic reduction is a consequence of the choice of primitives rather than of the intrinsic nature of monadic languages and that alternative formulations of the monadic syntax would not suffer from non-confluence.

The failure of confluence for ml-reduction requires a careful elaboration of the meta-theoretical results. The solution adopted in this paper is to 'complete' ml-reduction. Put it otherwise, we consider an extension $ml+$ of ml-reduction that is confluent and that has the same reflexive-symmetric-transitive closure as ml-reduction. An alternative would be to follow [18] and define a notion of erased pseudo-term and of erased reduction, but our approach has the advantage of being more concise.

Definition 14 The notion of κ-reduction is defined by the contraction rules

$$\mathsf{let}\ x{:}A \Leftarrow M\ \mathsf{in}\ N \to_\kappa \mathsf{let}\ x{:}\bullet \Leftarrow M\ \mathsf{in}\ N$$

The notion of $ml+$-reduction is defined as the union of ml and κ.

The notions of reduction ml and $ml+$ yield the same convertibility relation.

Lemma 16 $M =_{ml+} N \Leftrightarrow M =_{ml} N$

Proof Only the direct implication is interesting. To prove it, first show by induction on the structure of M that $M \to_\kappa N$ implies $M =_{ml} N$. Then proceed by induction on the derivation of $M =_{ml+} N$. ∎

Next we prove that $ml+$ is confluent. We start with some preliminary results.

Theorem 15 $\mathrm{SN}(\mu) = \mathcal{T}$.

Proof Using Finiteness of Developments. More precisely, define the set $\mathcal{U}$ of *underlined λ-terms* by the abstract syntax:

$$\mathcal{U} = \mathcal{V} \mid \bullet \mid \mathcal{U}\,\mathcal{U} \mid \lambda\mathcal{V}.\,\mathcal{U} \mid (\underline{\lambda}\mathcal{V}.\,\mathcal{U})\,\mathcal{U}$$

Then define the notion $\underline{\beta}$ of *underlined β-reduction* by the contraction rule

$$(\underline{\lambda}x.\,M)\,N \quad \to_{\underline{\beta}} \quad M\{x := N\}$$

and the notion of reduction $\underline{\beta}'$ by the contraction rule

$$P\,((\underline{\lambda}x.\,Q)\,R) \quad \to_{\underline{\beta}'} \quad (\underline{\lambda}x.\,P\,Q)\,R$$

provided $x \notin \mathrm{FV}(P)$. One possible formulation of Finiteness of Developments states $\mathcal{U} = \mathrm{SN}(\underline{\beta})$. Using an analysis of the possible infinite $\underline{\beta\beta'}$-reduction sequences, one can prove $\mathcal{U} = \mathrm{SN}(\underline{\beta\beta'})$, see Appendix. To prove our theorem, define a translation $[.] : \mathcal{T} \to \mathcal{U}$

as follows:

$$
\begin{array}{lll}
[x] & = x & x \in \mathcal{V} \\
[s] & = \bullet & s \in \mathcal{S} \\
[\lambda x{:}A.\, t] & = \mathsf{K}\ (\lambda x.\, [t])\ [A] & \\
[\Pi x{:}A.\, B] & = (\lambda x.\, [B])\ [A] & \\
[t\ u] & = \bullet\ [t]\ [u] & \\
[\mathsf{let}\ x{:}A \Leftarrow N\ \mathsf{in}\ M] & = (\underline{\lambda} x.\, \mathsf{K}\ [M]\ [A])\ [N] & \\
[\mathsf{unit}\ t] & = [t] & \\
[\mathsf{M}\ A] & = [A] & \\
[\mathsf{fix}\ x{:}A.\, M] & = \mathsf{K}\ (\lambda x.\, [M])\ [A] & \\
[\mathsf{nat}] & = \bullet & \\
[\lceil n \rceil] & = \bullet & \\
[\mathsf{succ}\ M] & = [M] & \\
[\mathsf{pred}\ M] & = [M] & \\
[\mathsf{if0}\ M\ M_1\ M_2] & = \bullet\ [M]\ [M_1]\ [M_2] &
\end{array}
$$

To conclude, we only need to show that [.] maps infinite sequences of μ-reductions are mapped into infinite sequences of $\underline{\beta\beta'}$-reductions, i.e. for every $P, Q \in \mathcal{T}$,

$$P \to_\mu Q \quad \Rightarrow \quad [P] \twoheadrightarrow^+_{\beta\beta'} [Q]$$

The latter is proved by induction on $M \in \mathcal{T}$. We treat three cases:

- $P \equiv \mathsf{let}\ x{:}A \Leftarrow (\mathsf{unit}\ N)\ \mathsf{in}\ M$ and $Q \equiv M\{x := N\}$.

$$
\begin{array}{rl}
[P] & \equiv\ (\underline{\lambda} x.\, \mathsf{K}\ [M]\ [A])\ [N] \\
& \to_{\underline{\beta}} (\underline{\lambda} x.\, [M])\ [N] \\
& \to_{\underline{\beta}} [M]\{x := [N]\} \\
& \equiv\ [M\{x := N\}] \\
& \equiv\ [Q]
\end{array}
$$

- $P \equiv \mathsf{let}\ x{:}A \Leftarrow M\ \mathsf{in}\ (\mathsf{unit}\ x)$ and $Q \equiv M$.

$$
\begin{array}{rl}
[P] & \equiv\ (\underline{\lambda} x.\, \mathsf{K}\ [x]\ [A])\ [M] \\
& \to_{\underline{\beta}} (\underline{\lambda} x.\, [x])\ [M] \\
& \to_{\underline{\beta}} [M] \\
& \equiv\ [Q]
\end{array}
$$

- $P \equiv \mathsf{let}\ x_2{:}A_2 \Leftarrow (\mathsf{let}\ x_1{:}A_1 \Leftarrow M_1\ \mathsf{in}\ M_2)\ \mathsf{in}\ M_3$ and $Q \equiv \mathsf{let}\ x_1{:}A_1 \Leftarrow M_1\ \mathsf{in}\ (\mathsf{let}\ x_2{:}A_2 \Leftarrow M_2\ \mathsf{in}\ M_3)$ assuming $x_1 \notin \mathrm{FV}(M_3) \cup \mathrm{FV}(A_2)$. We have

$$[P] \equiv (\underline{\lambda} x_2.\, \mathsf{K}\ [M_3]\ [A_2])\ ((\underline{\lambda} x_1.\, \mathsf{K}\ [M_2]\ [A_1])\ [M_1])$$

$$\begin{aligned}
&\to_{\underline{\beta}'} (\underline{\lambda} x_1.\ (\underline{\lambda} x_2.\ \mathsf{K}\ [M_3]\ [A_2])\ (\mathsf{K}\ [M_2]\ [A_1]))\ [M_1] \\
&\to_{\underline{\beta}'} (\underline{\lambda} x_1.\ \mathsf{K}\ ((\underline{\lambda} x_2.\ \mathsf{K}\ [M_3]\ [A_2])\ [M_2])\ [A_1])[M_1] \\
&\equiv\ (\underline{\lambda} x_1.\ \mathsf{K}\ [\mathsf{let}\ x_2{:}A_2 \Leftarrow M_2\ \mathsf{in}\ M_3]\ [A_1])\ [M_1] \\
&\equiv\ [\mathsf{let}\ x_1{:}A_1 \Leftarrow M_1\ \mathsf{in}\ (\mathsf{let}\ x_2{:}A_2 \Leftarrow M_2\ \mathsf{in}\ M_3)] \\
&\equiv\ [Q]
\end{aligned}$$

Other cases follow directly from the induction hypothesis. ■

Proposition 16 *$\mu\kappa$-reduction is confluent.*

Proof $\mu\kappa$-reduction is locally confluent, hence by Newman's Lemma it is enough to show that $\mathrm{SN}(\mu\kappa) = \mathcal{T}$. This may be proved by noting that:

(i) [.] maps κ-redexes to κ'-redexes where κ' is the reduction rule

$$M \to_{\kappa'} \bullet \qquad \text{if } M \not\equiv \bullet$$

(ii) κ' may be postponed and is strongly normalising.

■

Theorem 17 (Confluence) *ml+ is confluent.*

Theorem 17 is useful in proving several subsequent results, especially Subject Reduction. However, there does not seem to be any direct method from which to derive confluence of *ml*-reduction on legal terms, even if we assume the specification to be cubic. The situation is to be contrasted with that of PTSs with $\beta\eta$-conversion [18], where confluence of $\beta\eta$-reduction on legal terms is derived from normalisation and confluence. In our case we do not have *ml*-normalization so we cannot conclude.

4.4 Subject Reduction

Subject Reduction is a fundamental property of type systems: from a practical point of view, it ensures that types are closed under reduction. From a theoretical point of view, it is used throughout the metatheory of type systems, e.g. in consistency proofs and in operational semantics. As expected, the failure of confluence for *ml*-reduction yields slight complications in the proof of Subject Reduction. Nevertheless the proof of Subject Reduction for $\iota\phi$-reduction proceeds as usual.

Theorem 18 ($\iota\phi$-subject reduction) $\Gamma \vdash A : B,\ A \to_{\iota\phi} A' \quad \Rightarrow \quad \Gamma \vdash A' : B$

Proof Prove by simultaneous induction on the derivation of $\Gamma \vdash M : A$:

- $M \to_{\iota\phi} N \quad \Rightarrow \quad \Gamma \vdash N : A$
- if $\Gamma \to_{\iota\phi} \Delta \quad \Rightarrow \quad \Delta \vdash M : B$

(Recall that $\to_{\iota\phi}$ is extended to contexts by the clause $A \to_{\iota\phi} B \quad \Rightarrow \quad \Gamma, x : A, \Delta \to_{\iota\phi} \Gamma, x{:}B, \Delta$.) ■

The proof of Subject Reduction for $\beta\mu$-reduction requires a preliminary result, known as the Key Lemma, which follows directly from Theorem 17.

Lemma 17 (Key Lemma)

- *If* $\Pi x{:}A.\ B =_{ml} \Pi x{:}A'.\ B'$ *then* $A =_{ml} A'$ *and* $B =_{ml} B'$.
- *If* $\mathsf{M}\ A =_{ml} \mathsf{M}\ A'$ *then* $A =_{ml} A'$.

Proof The second step of both proofs follows from confluence of $\rightarrow_{ml+}$.

$$\begin{aligned}\Pi x{:}A.\ B =_{ml} \Pi x{:}C.\ D &\Rightarrow \Pi x{:}A.\ B =_{ml+} \Pi x{:}C.\ D \\ &\Rightarrow A =_{ml+} C \ \wedge\ B =_{ml+} D \\ &\Rightarrow A =_{ml} C \ \wedge\ B =_{ml} D\end{aligned}$$

$$\begin{aligned}\mathsf{M}\ A =_{ml} \mathsf{M}\ A' &\Rightarrow \mathsf{M}\ A =_{ml+} \mathsf{M}\ A' \\ &\Rightarrow A =_{ml+} A' \\ &\Rightarrow A =_{ml} A'\end{aligned}$$

■

Theorem 19 ($\beta\mu$-subject reduction) $\Gamma \vdash A : B,\ A \rightarrow_{\beta\mu} A' \quad \Rightarrow \quad \Gamma \vdash A' : B$

Proof Prove by simultaneous induction on the derivation of $\Gamma \vdash M : A$:

- $M \rightarrow_{\beta\mu} N \quad \Rightarrow \quad \Gamma \vdash N : A$
- if $\Gamma \rightarrow_{\beta\mu} \Delta \quad \Rightarrow \quad \Delta \vdash M : B$

(Recall that $\rightarrow_{\beta\mu}$ is extended to contexts by the clause $A \rightarrow_{\beta\mu} B \quad \Rightarrow \quad \Gamma, x : A, \Delta \rightarrow_{\beta\mu} \Gamma, x{:}B, \Delta$.) ■

The next result is a mild strengthening of the Key Lemma which is useful in the proof of strengthening.

Lemma 18

- *If* $s =_{ml} s'$ *then* $s \equiv s'$.
- *If* $A =_{ml} \mathsf{M}\ B$ *and* A *legal then* $A \twoheadrightarrow_{\beta} \mathsf{M}\ C$ *for some* $C =_{ml} B$.
- *If* $A =_{ml} s$ *and* A *legal then* $A \twoheadrightarrow_{\beta} s$.
- *If* $A =_{ml} \Pi x{:}B.\ C$ *and* A *legal then* $A \twoheadrightarrow_{\beta} \Pi x{:}D.\ E$ *with* $B =_{ml} D$ *and* $C =_{ml} E$.

An important consequence of Subject Reduction is strengthening, which is proved along the same lines as in [18].

Definition 20 **S** preserves sorts if

$$\Gamma \vdash A : s,\ \Gamma \vdash A' : s',\ A =_{ml} A' \quad \Rightarrow \quad s \equiv s'$$

The above technical assumption ensures that strengthening holds.

Theorem 21 *If* **S** *preserves sorts, then* **S** *satisfies strenghtening, i.e.*

$$\Gamma_1, x : A, \Gamma_2 \vdash b : B,\ x \notin \mathrm{FV}(\Gamma_2) \cup \mathrm{FV}(b) \cup \mathrm{FV}(B) \quad \Rightarrow \quad \Gamma_1, \Gamma_2 \vdash b : B$$

Proof First show that

$$\Gamma_1, x : A, \Gamma_2 \vdash b : B,\ x \notin \mathrm{FV}(\Gamma_2) \cup \mathrm{FV}(b) \cup \mathrm{FV}(B) \quad \Rightarrow \quad \exists B' \in \mathcal{T}.\, B =_{ml} B' \wedge \Gamma_1, \Gamma_2 \vdash b : B' \quad (*)$$

Then assume

$$\Gamma_1, x : A, \Gamma_2 \vdash b : B\ (\&),\ x \notin \mathrm{FV}(\Gamma_2) \cup \mathrm{FV}(b) \cup \mathrm{FV}(B)$$

Hence there exists $C =_{ml} B$ such that

$$\Gamma_1, \Gamma_2 \vdash b : C$$

By Correctness of Types on (&) we find that $B \in \mathcal{S}$ or $\Gamma_1, x : A, \Gamma_2 \vdash B : s$ for some $s \in \mathcal{S}$. In the second case, apply $(*)$ to find $E =_{ml} s$ such that

$$\Gamma_1, \Gamma_2 \vdash B : E$$

By Lemma 18 and Subject Reduction,

$$\Gamma_1, \Gamma_2 \vdash B : s$$

By (conversion)

$$\Gamma_1, \Gamma_2 \vdash b : B$$

In the first case, apply the Lemma 18 to conclude. In both cases, we obtain

$$\Gamma_1, \Gamma_2 \vdash b : B$$

To conclude the proof, we prove $(*)$ by induction on the structure of derivations.

- Assume the last rule is

$$\frac{\Gamma, y : C \vdash N : D \quad \Gamma \vdash (\Pi x{:}C.\, D) : s}{\Gamma \vdash \lambda y{:}C.\, N : \Pi y{:}C.\, D}$$

  with $\Gamma \equiv \Gamma_1, x : A, \Gamma_2$ and $x \notin \mathrm{FV}(\Gamma_2) \cup \mathrm{FV}(\lambda y{:}C.\, N) \cup \mathrm{FV}(\Pi y{:}C.\, D)$. By induction hypothesis, there exists E, F in $\mathcal{T}$ such that $E =_{ml} D$ and $F =_{ml} s$ and such that

$$\Gamma_1, \Gamma_2, y : C \vdash M : E$$

$$\Gamma_1, \Gamma_2 \vdash \Pi y{:}C.\, D : F$$

  By Generation, there exists $(s_1, s_2, s_3) \in \mathcal{R}$ such that

$$\Gamma_1, \Gamma_2 \vdash C : s_1$$

$$\Gamma_1, \Gamma_2, y : C \vdash D : s_2$$

  By Correctness of Types, there exists $s' \in \mathcal{S}$ such that

$$\Gamma_1, \Gamma_2, y : C \vdash E : s'$$

By Preservation of Sorts, $s' \equiv s_2$ and hence by (product)

$$\Gamma_1, \Gamma_2 \vdash \Pi y{:}C.\, D : s_3$$

By (abstraction),

$$\Gamma_1, \Gamma_2 \vdash \lambda y{:}C.\, M : \Pi y{:}C.\, D$$

- Assume the last rule is

$$\frac{\Gamma \vdash M : \mathsf{M}\, C \quad \Gamma, y : C \vdash N : \mathsf{M}\, D}{\Gamma \vdash \mathsf{let}\ y{:}C \Leftarrow M\ \mathsf{in}\ N : \mathsf{M}\, D}$$

with $\Gamma \equiv \Gamma_1, x : A, \Gamma_2$ and $x \notin \mathrm{FV}(\Gamma_2) \cup \mathrm{FV}(\mathsf{let}\ y{:}C \Leftarrow M\ \mathsf{in}\ N) \cup \mathrm{FV}(D)$. By induction hypothesis, there exists E, F in $\mathcal{T}$ such that $E =_{ml} \mathsf{M}\, C$ and $F =_{ml} \mathsf{M}\, D$ and such that

$$\Gamma_1, \Gamma_2 \vdash M : E$$

$$\Gamma_1, \Gamma_2, y : C \vdash N : F$$

By Correctness of types, $\Gamma_1, \Gamma_2 \vdash C : s$. By Extended Uniqueness of Types $s =_{ml} *^x$. By Lemma 18, $s \equiv *^x$ and hence by (monad) $\Gamma_1, \Gamma_2 \vdash \mathsf{M}\, C : *^c$. By (conversion),

$$\Gamma_1, \Gamma_2 \vdash M : E$$

By Lemma 18, there exists $G \in \mathcal{T}$ such that $F \twoheadrightarrow_\beta \mathsf{M}\, G$. By β-Subject Reduction,

$$\Gamma_1, \Gamma_2, y : C \vdash N : \mathsf{M}\, G$$

By (let),

$$\Gamma_1, \Gamma_2 \vdash \mathsf{let}\ y{:}C \Leftarrow M\ \mathsf{in}\ N : \mathsf{M}\, G$$

- Assume the last rule is

$$\frac{\Gamma, y : \mathsf{M}\, B \vdash M : \mathsf{M}\, B}{\Gamma \vdash \mathsf{fix}\ y{:}\mathsf{M}\, B.\, M : \mathsf{M}\, B}$$

with $\Gamma \equiv \Gamma_1, x : A, \Gamma_2$ and $x \notin \mathrm{FV}(\Gamma_2) \cup \mathrm{FV}(\mathsf{fix}\ y{:}\mathsf{M}\, B.\, M)$. By induction hypothesis, there exists $E \in \mathcal{T}$ such that $E =_{ml} \mathsf{M}\, B$ and

$$\Gamma_1, \Gamma_2, y : \mathsf{M}\, B \vdash M : E$$

By Correctness of Types and Weakening,

$$\Gamma_1, \Gamma_2 \vdash \mathsf{M}\, B : *^c$$

By (conversion),

$$\Gamma_1, \Gamma_2, y : \mathsf{M}\, B \vdash M : \mathsf{M}\, B$$

and by (fixpoint),

$$\Gamma_1, \Gamma_2 \vdash \mathsf{fix}\ y{:}\mathsf{M}\, B.\, M : \mathsf{M}\, B$$

■

As we shall see later, most cubic specifications preserve sorts.

4.5 Uniqueness of Types and Classification

This Subsection is concerned with properties that hold for some specific classes of MTSs. The first class we consider is that of functional specifications.

Definition 22 A specification $\mathbf{S} = (\mathcal{S}, *^v, *^c, \mathcal{A}, \mathcal{R})$ is *functional* if for every $s_1, s_2, s_2', s_3, s_3' \in \mathcal{S}$,

$$(s_1, s_2) \in \mathcal{A} \quad \wedge (s_1, s_2') \in \mathcal{A} \quad \Rightarrow s_2 \equiv s_2'$$
$$(s_1, s_2, s_3) \in \mathcal{R} \wedge (s_1, s_2, s_3') \in \mathcal{R} \Rightarrow s_3 \equiv s_3'$$

Functional specifications enjoy uniqueness of types.

Lemma 19 *If* $\Gamma \vdash M : A$ *and* $\Gamma \vdash M : B$ *then* $A =_{ml} B$.

Proof By induction on the structure of derivations. ■

Uniqueness of Types may be used to prove preservation of sorts.

Lemma 20 *If* $\mathbf{S}$ *is cubic and functional then it preserves sorts.*

Proof It is enough to show that for every A, A' in β-normal form, we have

$$\Gamma \vdash A : s,\ \Gamma \vdash A' : s',\ A =_{ml} A' \quad \Rightarrow \quad s \equiv s'$$

Types in β-normal form are given by the syntax

$$\mathcal{U} = \mathcal{S} \mid \mathcal{B} \mid \Pi \mathcal{V}{:}\mathcal{U}.\,\mathcal{U} \mid \mathsf{M}\,\mathcal{U} \mid \mathsf{nat}$$

where

$$\mathcal{B} = \mathcal{V} \mid \mathcal{V}\,\mathcal{T}$$

We now reason by induction on the definition of $\mathcal{U}$, using Uniqueness of Types. ■

We now turn to Classification. For PTSs the result is traditionally expressed for the so-called injective specifications and states—among other things—that for every $M \in \mathcal{T}$ and $s, s' \in \mathcal{S}$

$$\Gamma \vdash M : A : s \quad \wedge \quad \Gamma' \vdash M : A' : s' \quad \Rightarrow \quad s \equiv s'$$

One can prove a similar result for cubic MTSs—using preservation of sorts. However several MTSs of interest are not cubic and the statement of Classification itself is overly strong for the purpose of our operational semantics. What is needed is a stratification of terms in the usual three layers: objects, types and kinds—we only focus on cubic specifications. [7]

[7] Note that one may probably relax the assumption $\mathcal{S}^{\perp} = \{*^v, *^c\}$ in the statement of the Lemma.

Lemma 21 (Classification) *Let* $\mathbf{S} = (\mathcal{S}, *^v, *^c, \mathcal{A}, \mathcal{R})$ *be a cubic specification such that* $\mathcal{S}^\perp = \{*^v, *^c\}$. *Let* $*$ *range over* $\mathcal{S}^\perp$ *and* $\Box$ *range over* $\mathcal{S}^\top$. *Define*

$$\begin{aligned}
\mathcal{O} &= \mathcal{V}^* \mid \mathcal{O}\,\mathcal{O} \mid \mathcal{O}\,\mathcal{C} \mid \lambda\mathcal{V}^*{:}\mathcal{C}.\,\mathcal{O} \mid \lambda\mathcal{V}^\Box{:}\mathcal{K}.\,\mathcal{O} \mid \lceil 0 \rceil \mid \mathsf{succ}\ \mathcal{O} \mid \mathsf{pred}\ \mathcal{O} \mid \mathsf{if0}\ \mathcal{O}\ \mathcal{O}\ \mathcal{O} \mid \\
&\quad \mathsf{unit}\ \mathcal{O} \mid \mathsf{let}\ \mathcal{V}^*{:}\mathcal{C} \Leftarrow \mathcal{O}\ \mathsf{in}\ \mathcal{O} \mid \mathsf{fix}\ \mathcal{V}^*{:}\mathcal{C}.\,\mathcal{O} \\
\mathcal{C} &= \mathcal{V}^\Box \mid \mathcal{C}\,\mathcal{O} \mid \mathcal{C}\,\mathcal{C} \mid \lambda\mathcal{V}^*{:}\mathcal{C}.\,\mathcal{C} \mid \lambda\mathcal{V}^\Box{:}\mathcal{K}.\,\mathcal{C} \mid \Pi\mathcal{V}^*{:}\mathcal{C}.\,\mathcal{C} \mid \Pi\mathcal{V}^\Box{:}\mathcal{K}.\,\mathcal{C} \mid \mathsf{nat} \mid \mathsf{M}\ \mathcal{C} \\
\mathcal{K} &= * \mid \Pi\mathcal{V}^*{:}\mathcal{C}.\,\mathcal{K} \mid \Pi\mathcal{V}^\Box{:}\mathcal{K}.\,\mathcal{K}
\end{aligned}$$

Then $\mathcal{O}, \mathcal{C}, \mathcal{K}$ *are pairwise disjoint. Moreover,*

(i) If $\Gamma \vdash M : A : *$ *then* $M \in \mathcal{O}$;

(ii) If $\Gamma \vdash M : A : \Box$ *then* $M \in \mathcal{C}$;

(iii) If $\Gamma \vdash M : \Box$ *then* $M \in \mathcal{K}$.

Proof By using properties of $\lceil . \rceil$ and a similar classification result for the Calculus of Constructions with fixpoints. ■

It follows from Subject Reduction that for every M, N in $\mathcal{T}$ such that M is legal and $M \twoheadrightarrow_{ml} N$ that

$$M \in \mathcal{X} \quad \Rightarrow \quad N \in \mathcal{X}$$

for $\mathcal{X}$ ranging over $\mathcal{O}, \mathcal{C}, \mathcal{K}$.

4.6 Type-checking

Type-checking is in general undecidable because we cannot check convertibility of types. However, type-checking becomes decidable if types terminate.

Definition 23 A cubic specification $\mathbf{S} = (\mathcal{S}, *^v, *^c, \mathcal{A}, \mathcal{R})$ is *non-dependent* if for every $(s_1, s_2, s_3) \in \mathcal{R}$,

$$s_1 \in \mathcal{S}^\perp \quad \Rightarrow \quad s_2, s_3 \in \mathcal{S}^\perp$$

Non-dependent cubic specifications have normalising types, which is the key to decidable type-checking.

Proposition 24 *Non-dependent functional cubic specifications have decidable type-checking provided* $\mathcal{S}, \mathcal{A}, \mathcal{R}$ *are recursive.*

5 Operational Theory

In this section, we give an operational theory for the class of MTS specifications that we consider to be *strictly monadic*. The goal is to obtain an appropriate notion of MTSprogram equivalence. A general operational theory that allowed for an arbitrary monad would quite difficult to develop. As a first step, we consider only the computational effect of non-termination as expressed by the lifting monad.

The foundation of this material is Gordon's work [21] on an operational theory for a simply-typed version of the computational with inductive and co-inductive types, and

many of our definitions and properties are minor adaptations of Gordon's. The main technical challenges in scaling up Gordon's work are dealing with conversion in types in kinds and handling dependent types.

- *Conversion in types:* In Gordon's presentation, the notion of a type-indexed relation is used in many places. With conversion in types, this becomes more complicated. For example, one must typically show that the type-indexed relations in our setting are closed under conversion of types, and substitution of terms in types.
- *Dependent types:* With dependent type, objects may occur in types and thus many notions regarding substitution and contexts become more complicated.

5.1 Program evaluation

With the single syntactic category of pseudo-terms in MTSs (as adapted from PTSs), it is not immediately clear how to define a notion of evaluation. Classes of terms that we intuitively understand to be objects, types, and kinds all belong to the same set. Normally, only objects (which express computation) should be evaluated.

Given this mixing of terms with clearly different roles, we feel that is it important to impose order by considering systems arising from cubic specifications. Lemma 21 (classification) guarantees that terms in cubic specifications can be separated into disjoint classes of objects, type constructors, and kinds.

Even with this restriction, dependent types introduce complications. With dependent types, objects can occur in types, and thus type-checking inevitably requires some notion of evaluation. This well-known phenomenon has been described as a loss of distinction between phases of type-checking and evaluation [8].

Our approach to handling this situation is to simply define a notion of evaluation for well-typed objects resulting from cubic specifications. Specifically, we assume that type-checking (by whatever means) has already been performed. Thus, for example, evaluation never reduces type arguments to polymorphic abstractions, nor does it evaluate terms that appear in types. Any such manipulation would be performed by type-checking. The classification implied by cubic specifications along with subject reduction guarantees that the class of objects is closed under evaluation — that is, evaluation of objects need only involve objects.

We now turn to the following definition specifying the notions of *program* and *program reduction* and *evaluation.*

Definition 25

- A *program* is a closed object M with a closed type, *i.e.*, it is an object M where there exists an A such that $\cdot \vdash M : A : *^x$ for $x \in \{v, c\}$.
- A *value* is an object of given by the following grammar:

  $V ::= \lceil n \rceil \mid \lambda x{:}A.\, M \mid \mathsf{unit}\ M.$
- An *experiment* is an object context (with exactly one hole) given by the following grammar:

  $E ::= \mathsf{succ}\ \bullet \mid \mathsf{pred}\ \bullet \mid \mathsf{if0}\ \bullet\ M\ N \mid \bullet\ N \mid \mathsf{let}\ x{:}A \Leftarrow \bullet\ \mathsf{in}\ M$

$$\mathsf{succ}\ \lceil n \rceil \longmapsto_{ml} \lceil n+1 \rceil$$

$$\mathsf{pred}\ \lceil n+1 \rceil \longmapsto_{ml} \mathsf{unit}\ \lceil n \rceil$$

$$\mathsf{pred}\ \lceil 0 \rceil \longmapsto_{ml} \mathsf{unit}\ \lceil 0 \rceil$$

$$\mathsf{if0}\ \lceil 0 \rceil\ M_2\ M_3 \longmapsto_{ml} M_2$$

$$\mathsf{if0}\ \lceil n+1 \rceil\ M_2\ M_3 \longmapsto_{ml} M_3$$

$$(\lambda x{:}A.\ M_0)\ M_1 \longmapsto_{ml} M_0\{x := M_1\}$$

$$\mathsf{fix}\ x{:}A.\ M \longmapsto_{ml} M\{x := \mathsf{fix}\ x{:}A.\ M\}$$

$$\mathsf{let}\ x{:}A \Leftarrow \mathsf{unit}\ M_1\ \mathsf{in}\ N_2 \longmapsto_{ml} M_2\{x := M_1\}$$

$$\frac{e \longmapsto_{ml} e'}{E[e] \longmapsto_{ml} E[e']}$$

Fig. 17. Program reduction rules

$$\lceil n \rceil \Downarrow \lceil n \rceil \qquad \lambda x{:}A.\ M \Downarrow \lambda x{:}A.\ M \qquad \mathsf{unit}\ M \Downarrow \mathsf{unit}\ M$$

$$\mathsf{succ}\ \lceil n \rceil \Downarrow \lceil n+1 \rceil \qquad \mathsf{pred}\ \lceil n+1 \rceil \Downarrow \lceil n \rceil \qquad \mathsf{pred}\ \lceil 0 \rceil \Downarrow \lceil 0 \rceil$$

$$\frac{M_2 \Downarrow V}{\mathsf{if0}\ \lceil 0 \rceil\ M_2\ M_3 \Downarrow V} \qquad \frac{M_3 \Downarrow V}{\mathsf{if0}\ \lceil n+1 \rceil\ M_2\ M_3 \Downarrow V}$$

$$\frac{M_0 \Downarrow \lambda x{:}A.\ M_0' \qquad M_0'\{x := M_1\} \Downarrow V}{M_0\ M_1 \Downarrow V} \qquad \frac{M\{x := \mathsf{fix}\ x{:}A.\ M\} \Downarrow V}{\mathsf{fix}\ x{:}A.\ M \Downarrow V}$$

$$\frac{M \Downarrow \mathsf{unit}\ M' \qquad N\{x := M'\} \Downarrow V}{\mathsf{let}\ x{:}A \Leftarrow M\ \mathsf{in}\ N \Downarrow V}$$

Fig. 18. Program evaluation rules

- A *confined object* is a triple (Γ, M, A) such that M is an object and $\Gamma \vdash M : A$ is derivable
- *Single-step program reduction* $\cdot \longmapsto_{ml} \cdot$ is the smallest relation on programs satisfying the rules of Figure 17.
- *Big-step program evaluation* $\cdot \Downarrow \cdot$ is the smallest relation on programs satisfying the rules of Figure 18.

We write $M \Downarrow$ when there exists a V such that $M \Downarrow V$ and $M \Uparrow$ when there does not exist a V such that $M \Downarrow V$.

5.2 Strictly monadic specifications

A fundamental property of monadic frameworks is that canonical programs have value type. Above we noted that canonical programs include abstractions — and thus we desire that they have a value type (*i.e.*, they belong to El^{*^v}). To ensure this, one needs to consider a restriction on specifications that forces object abstractions to be typed as values.

Definition 26 A specification $\mathcal{R}$ is *value adhering* if $s' = *^v$ for every $(s, *^x, s') \in \mathcal{R}$.

We have considered several restrictions on specifications, and we now give a final restriction that captures what we feel are the natural requirements for a specification to have truly monadic character.

Definition 27 A specification $\mathcal{R}$ is *strictly monadic* if it is

- functional,
- cubic and $\mathcal{S}^\perp = \{*^v, *^c\}$, and
- value-adhering.

Thus, a strictly monadic specification guarantees uniqueness of types, ensures a three-level stratification into objects, constructors, and kinds, and forces object abstractions to be values (observe that if $\mathcal{R}$ is a strictly monadic specification and $M\ N \in \mathsf{El}^{*^v}$, then $M \in \mathsf{El}^{*^v}$)

Throughout the remainder of this section, we will let $*$ range over bottom sorts $\mathcal{S}^\perp$ and $\Box$ range over top sorts $\mathcal{S}^\top$.

We now give several basic properties of evaluation under strictly monadic specifications.

Proposition 28 *Assume that $\mathcal{R}$ is strictly monadic. Let L, M, N be programs and U, V be values.*

(i) Program reduction is deterministic: *if* $L \longmapsto_{ml} M$ *and* $L \longmapsto_{ml} N$ *then* $M \equiv N$.

(ii) Program evaluation is deterministic: *if* $L \Downarrow U$ *and* $L \Downarrow V$ *then* $U \equiv V$.

(iii) Values are the canonical forms of program reduction: *if* $\neg\exists N$ *such* $M \longmapsto_{ml} N$, *then* M *is a value.*

(iv) Suppose $M \longmapsto_{ml} N$. *Then for any* V, $N \Downarrow V$ *implies* $M \Downarrow V$.

(v) $M \Downarrow V$ *iff* $M \longmapsto^*_{ml} V$

Proof Follows the same structure as the proof by Gordon [21]. ∎

Proposition 29 *Assume that $\mathcal{R}$ is strictly monadic.*

(i) If $\vdash A : *^v$ *then* $A \twoheadrightarrow_{ml} \mathsf{nat}$, *or* $A \twoheadrightarrow_{ml} \Pi x{:}B.\, C$ *for some* $B, C \in \mathcal{T}$.

(ii) If $\vdash A : *^c$ *then* $A \twoheadrightarrow_{ml} \mathsf{M}\ B$ *for some* $B \in \mathcal{T}$.

Proof Proof of (1); Necessarily A is β-strongly normalizing. Let A' be its β-normal form. By Subject Reduction $\vdash A' : *^v$. By Generation and strict monadicity, A' can only be nat or a product so we are done. The proof for (2) is similar. ■

The following property formalizes the notion that programs of value type must converge to a canonical program.

Proposition 30 *Assume that $\mathcal{R}$ is strictly monadic.*

(i) If $\vdash M : A : *^v$ then $M \Downarrow$.

(ii) If $\vdash M : \mathsf{nat}$, then $M \Downarrow \lceil n \rceil$.

(iii) If $\vdash M : \Pi x{:}A.\, B : *^v$ then $M \Downarrow \lambda x{:}A'.\, N$.

Proof (1) By a case analysis on the possible shapes of M, conclude that every weak-head reduction sequence starting from M is a $\beta\iota$-reduction sequence. Hence (by strong normalization of $\beta\iota$-reduction) the weak-head reduction sequence starting from M must end. (2) and (3) Again, case analysis on the possible shapes of M. ■

Proposition 31 (program type inhabitation) *Assume that $\mathcal{R}$ is strictly monadic.*

(i) There is no pseudo-term M s.t. $\cdot \vdash M : \Pi X{:}*^v.\, X$. Thus, some value program types are uninhabited.

(ii) For any A such that $\cdot \vdash A : *^c$, there is a B such that $\cdot \vdash \mathsf{fix}\ x{:}\mathsf{M}\ B.\, x : A$. Thus, all computation program types are inhabited.

Proof (1) Without loss of generality we may assume that M is in β-normal form. We proceed by a case analysis on the possible shapes of M.

(2) We know $A \twoheadrightarrow_{ml} \mathsf{M}\ B$ for some $B \in \mathcal{T}$. By Subject Reduction, $\vdash \mathsf{M}\ B : *^c$. By (start), $x : \mathsf{M}\ B \vdash x : \mathsf{M}\ B$. By (fixpoint), $\vdash \mathsf{fix}\ x{:}\mathsf{M}\ B.\, x : \mathsf{M}\ B$. By (conversion), $\vdash \mathsf{fix}\ x{:}\mathsf{M}\ B.\, x : A$. ■

Let $\bot_A$ represent the program $\mathsf{fix}\ x{:}\mathsf{M}\ B.\, x$ of type A that exists (as shown above) for any $A \in \mathcal{T}$ such that $\vdash A : *^c$. We write $\bot$ when the type A is clear from the context.

5.3 Operational equality

We now establish a notion of operational equivalence *applicative bisimilarity* for strictly monadic MTSs following Gordon's elegant presentation of a similar notion for a simply-typed version of the computational metalanguage with inductive/co-inductive types [21]. Given the definitions below, Gordon's proofs [21] carry over in a reasonably straightforward manner. Only issues regarding substitution and type equivalence are more complicated, and we indicate some of the interesting points below. In the end, we can show that the MTS calculus is sound w.r.t. our notion of operational equivalence.

Definition 32

- A *ground relation* R is a binary relation between programs of the same type.
- A *confined relation* is a binary relation between confined objects of the same type in context Γ. Write $\Gamma \vdash M\,\mathcal{R}\,N : A$ to mean that $(\Gamma \vdash M : A)\,\mathcal{R}\,(\Gamma \vdash N : A)$.

Definition 33 A ground relation $\mathcal{R}$ is *operationally adequate* iff for all programs M,N, and canonical programs V where $\cdot \vdash M, N, V : A$:

(i) If $M \longmapsto_{ml} N$ then $N \mathcal{R} M$.

(ii) If $M \Downarrow V$ then $V \mathcal{R} M$.

(iii) $M \Uparrow$ iff $M \mathcal{R} \bot_A$.

(iv) $M \Downarrow$ iff for some V, $V \mathcal{R} M$.

We often need to consider confined relations that are *compatible* with the syntactic structure of objects. Figure 19 introduces a special notion for this property of confined relations called *compatible closure.* All the rules have an implicit side condition that any sentence denoting a pair of confined terms is well-formed. Specifically, a sentence $\Gamma \vdash M \mathcal{R} N : A$ is well-formed if $\Gamma \vdash M, N : A$. Thus, the rule

$$\frac{\Gamma \vdash M_1 \mathcal{R} N_1 : \Pi x{:}A.\, B \qquad \Gamma \vdash M_2 \mathcal{R} N_2 : A}{\Gamma \vdash (M_1\, M_2)\, \widehat{\mathcal{R}}\, (N_1\, N_2) : B\{x := M_2\}}$$

should be read as follows: if $\Gamma \vdash M_1, N_1 : \Pi x{:}A.\, B$ and $\Gamma \vdash M_1 \mathcal{R} N_1 : \Pi x{:}A.\, B$ and $\Gamma \vdash M_2, N_2 : A$, then if $\Gamma \vdash M_1\, M_2, N_1\, N_2 : B\{x := M_2\}$ then $\Gamma \vdash (M_1\, M_2)\, \widehat{\mathcal{R}}\, (N_1\, N_2) : B\{x := M_2\}$. Note that the MTS typing rules also imply $\Gamma \vdash N_1\, N_2 : B\{x := N_2\}$. Then, by uniqueness of types we have $B\{x := M_2\} =_{ml} B\{x := N_2\}$. Thus, by the conversion rule for compatible closure we have $\Gamma \vdash (M_1\, M_2)\, \widehat{\mathcal{R}}\, (N_1\, N_2) : B\{x := N_2\}$.

To clarify such properties, the we now give a proposition regarding compatible closure that is analogous to the Generation Lemma (Lemma 14) for MTS typing judgements.

Proposition 34 *Let $\mathcal{R}$ be a confined relation, and consider judgements of the form $\Gamma \vdash M \widehat{\mathcal{R}} N : A$.*

- *If* $\Gamma \vdash \lceil n \rceil\, \widehat{\mathcal{R}}\, \lceil n \rceil : A$,
  then $\Gamma \vdash \lceil n \rceil\, \widehat{\mathcal{R}}\, \lceil n \rceil : \mathsf{nat}$ *and* $A =_{ml} \mathsf{nat}$
- *If* $\Gamma \vdash (\mathsf{succ}\, M)\, \widehat{\mathcal{R}}\, (\mathsf{succ}\, N) : A$,
  then $\Gamma \vdash (\mathsf{succ}\, M)\, \widehat{\mathcal{R}}\, (\mathsf{succ}\, N) : \mathsf{nat}$, *and* $A =_{ml} \mathsf{nat}$
  and $\Gamma \vdash M \mathcal{R} N : \mathsf{nat}$
- *If* $\Gamma \vdash (\mathsf{pred}\, M)\, \widehat{\mathcal{R}}\, (\mathsf{pred}\, N) : A$,
  then $\Gamma \vdash (\mathsf{pred}\, M)\, \widehat{\mathcal{R}}\, (\mathsf{pred}\, N) : \mathsf{nat}$, *and* $A =_{ml} \mathsf{nat}$
  and $\Gamma \vdash M \mathcal{R} N : \mathsf{nat}$
- *If* $\Gamma \vdash (\mathsf{if0}\, M_1\, M_2\, M_3)\, \widehat{\mathcal{R}}\, (\mathsf{if0}\, N_1\, N_2\, N_3) : A$,
  then $\Gamma \vdash (\mathsf{if0}\, M_1\, M_2\, M_3)\, \widehat{\mathcal{R}}\, (\mathsf{if0}\, N_1\, N_2\, N_3) : \mathsf{M}\, B$ *for some* B
  and $A =_{ml} \mathsf{M}\, B$
  and $\Gamma \vdash M_1 \mathcal{R} N_1 : \mathsf{nat}$, $\Gamma \vdash M_2 \mathcal{R} N_2 : \mathsf{M}\, B$, *and* $\Gamma \vdash M_3 \mathcal{R} N_3 : \mathsf{M}\, A$.

$$\Gamma \vdash x \,\widehat{\mathcal{R}}\, x : A \qquad \Gamma \vdash \lceil n \rceil \,\widehat{\mathcal{R}}\, \lceil n \rceil : \mathsf{nat}$$

$$\frac{\Gamma \vdash M \,\mathcal{R}\, N : \mathsf{nat}}{\Gamma \vdash (\mathsf{succ}\, M) \,\widehat{\mathcal{R}}\, (\mathsf{succ}\, N) : \mathsf{nat}} \qquad \frac{\Gamma \vdash M \,\mathcal{R}\, N : \mathsf{nat}}{\Gamma \vdash (\mathsf{pred}\, M) \,\widehat{\mathcal{R}}\, (\mathsf{pred}\, N) : \mathsf{nat}}$$

$$\frac{\Gamma \vdash M_1 \,\mathcal{R}\, N_1 : \mathsf{nat} \quad \Gamma \vdash M_2 \,\mathcal{R}\, N_2 : \mathsf{M}\, A \quad \Gamma \vdash M_3 \,\mathcal{R}\, N_3 : \mathsf{M}\, A}{\Gamma \vdash (\mathsf{if0}\, M_1\, M_2\, M_3) \,\widehat{\mathcal{R}}\, (\mathsf{if0}\, N_1\, N_2\, N_3) : \mathsf{M}\, A}$$

$$\frac{\Gamma, x{:}A \vdash M \,\mathcal{R}\, N : B}{\Gamma \vdash (\lambda x{:}A_1.\, M) \,\widehat{\mathcal{R}}\, (\lambda x{:}A_2.\, N) : \Pi x{:}A.\, B}$$

$$\frac{\Gamma, x{:}\mathsf{M}\, A \vdash M \,\mathcal{R}\, N : \mathsf{M}\, A}{\Gamma \vdash (\mathsf{fix}\, x{:}A_1.\, M) \,\widehat{\mathcal{R}}\, (\mathsf{fix}\, x{:}A_2.\, N) : \mathsf{M}\, A}$$

$$\frac{\Gamma \vdash M_1 \,\mathcal{R}\, N_1 : \Pi x{:}A.\, B \quad \Gamma \vdash M_2 \,\mathcal{R}\, N_2 : A}{\Gamma \vdash (M_1\, M_2) \,\widehat{\mathcal{R}}\, (N_1\, N_2) : B\{x := M_2\}} \text{ if } \Gamma \vdash A : *$$

$$\frac{\Gamma \vdash M_1 \,\mathcal{R}\, N_1 : \Pi x{:}A.\, B}{\Gamma \vdash (M_1\, A_1) \,\widehat{\mathcal{R}}\, (N_1\, A_2) : B\{x := A_1\}} \text{ if } \Gamma \vdash A : \Box$$

$$\frac{\Gamma \vdash M \,\mathcal{R}\, N : A}{\Gamma \vdash (\mathsf{unit}\, M) \,\widehat{\mathcal{R}}\, (\mathsf{unit}\, N) : \mathsf{M}\, A}$$

$$\frac{\begin{array}{c}\Gamma \vdash M_1 \,\mathcal{R}\, N_1 : \mathsf{M}\, A \\ \Gamma, x{:}A \vdash M_2 \,\mathcal{R}\, N_2 : \mathsf{M}\, B \\ \Gamma \vdash A_i : * \quad A_i =_{ml} \mathsf{M}\, A \text{ for } i = 1, 2\end{array}}{\Gamma \vdash (\mathsf{let}\, x{:}A_1 \Leftarrow M_1\, \mathsf{in}\, M_2) \,\widehat{\mathcal{R}}\, (\mathsf{let}\, x{:}A_2 \Leftarrow N_1\, \mathsf{in}\, N_2) : \mathsf{M}\, B}$$

$$\frac{\Gamma \vdash M \,\mathcal{R}\, N : A \quad \Gamma \vdash A' : s \quad A =_{ml} A'}{\Gamma \vdash M \,\widehat{\mathcal{R}}\, N : A'}$$

Fig. 19. Compatible closure of a confined relation

- *If* $\Gamma \vdash (\lambda x{:}A_1.\, M)\,\widehat{\mathcal{R}}\,(\lambda x{:}A_2.\, N) : A$,
  then $\Gamma \vdash (\lambda x{:}A_1.\, M)\,\widehat{\mathcal{R}}\,(\lambda x{:}A_2.\, N) : \Pi x{:}B.\, C$ *for some* B *and* C,
  and $A =_{ml} \Pi x{:}B.\, C$, $A_1 =_{ml} B$, *and* $A_2 =_{ml} B$,
  and $\Gamma, x{:}B \vdash M \mathcal{R} N : C$.

- *If* $\Gamma \vdash (\mathsf{fix}\ x{:}A_1.\, M)\,\widehat{\mathcal{R}}\,(\mathsf{fix}\ x{:}A_2.\, N) : A$,
  then $\Gamma \vdash (\mathsf{fix}\ x{:}A_1.\, M)\,\widehat{\mathcal{R}}\,(\mathsf{fix}\ x{:}A_2.\, N) : \mathsf{M}\ B$ *for some* B,
  and $A =_{ml} \mathsf{M}\ B$, $A_1 =_{ml} \mathsf{M}\ B$, *and* $A_2 =_{ml} \mathsf{M}\ B$,
  and $\Gamma, x{:}\mathsf{M}\ B \vdash M \mathcal{R} N : \mathsf{M}\ B$.

- *If* $\Gamma \vdash (M_1\ M_2)\,\widehat{\mathcal{R}}\,(N_1\ N_2) : A$,
  then *one of the following holds:*

(i) $\Gamma \vdash (M_1\ M_2)\,\widehat{\mathcal{R}}\,(N_1\ N_2) : B\{x := M_2\}$ *for some* B,
and $A =_{ml} B\{x := M_2\}$ *and* $A =_{ml} B\{x := N_2\}$,
and $\Gamma \vdash M_1 \mathcal{R} N_1 : \Pi x{:}C.\, B$ *and* $\Gamma \vdash M_2 \mathcal{R} N_2 : C$,
and $\Gamma \vdash C : *$.

(ii) $\Gamma \vdash (M_1\ M_2)\,\widehat{\mathcal{R}}\,(N_1\ N_2) : B\{x := M_2\}$ *for some* B,
and $A =_{ml} B\{x := M_2\}$ *and* $A =_{ml} B\{x := N_2\}$,
and $\Gamma \vdash M_1 \mathcal{R} N_1 : \Pi x{:}C.\, B$,
and $\Gamma \vdash C : \Box$.

- *If* $\Gamma \vdash (\mathsf{unit}\ M)\,\widehat{\mathcal{R}}\,(\mathsf{unit}\ N) : A$,
  then $\Gamma \vdash (\mathsf{unit}\ M)\,\widehat{\mathcal{R}}\,(\mathsf{unit}\ N) : \mathsf{M}\ B$ *and* $A =_{ml} \mathsf{M}\ B$
  and $\Gamma \vdash M \mathcal{R} N : B$.

- *If* $\Gamma \vdash (\mathsf{let}\ x{:}A_1 \Leftarrow M_1\ \mathsf{in}\ M_2)\,\widehat{\mathcal{R}}\,(\mathsf{let}\ x{:}A_2 \Leftarrow N_1\ \mathsf{in}\ N_2) : A$
  then $\Gamma \vdash (\mathsf{let}\ x{:}A_1 \Leftarrow M_1\ \mathsf{in}\ M_2)\,\widehat{\mathcal{R}}\,(\mathsf{let}\ x{:}A_2 \Leftarrow N_1\ \mathsf{in}\ N_2) : \mathsf{M}\ C$
  and $A =_{ml} \mathsf{M}\ C$
  and $\Gamma \vdash M_1 \mathcal{R} N_1 : \mathsf{M}\ B$ *and* $\Gamma, x{:}B \vdash M_2 \mathcal{R} N_2 : \mathsf{M}\ C$
  and $\Gamma \vdash A_i : *$ *and* $A_i =_{ml} B$ *for* $i = 1, 2$.

Proof By induction of the derivation of $\Gamma \vdash M\,\widehat{\mathcal{R}}\,N : A$. ∎

MTS judgements $\Gamma \vdash M : A$ have various structural properties such as thinning, condensing, *etc.* that were presented earlier. The definition below gives analogous properties

for confined relations, and we will need to verify that the confined relations we introduce do indeed satisfy one or more of these properties (we are not always need them to possess all of properties).

Definition 35 (Properties of confined relations)

(i) **Type Conversion:**

$$\text{If } \Gamma \vdash M \mathcal{R} N : A \quad \text{and} \quad \Gamma \vdash A' : s$$
$$\text{such that } A =_{ml} A'$$
$$\text{then } \Gamma \vdash M \mathcal{R} N : A'$$

(ii) **Context Conversion:**

$$\text{If } \Gamma \vdash M \mathcal{R} N : A \quad \text{and} \quad \Gamma' \text{ is a valid context}$$
$$\text{such that } \Gamma =_{ml} \Gamma'$$
$$\text{then } \Gamma' \vdash M \mathcal{R} N : A$$

(iii) **Thinning:** Let Γ and Δ be legal contexts such that $\Gamma \subseteq \Delta$. Then

$$\Gamma \vdash M \mathcal{R} N : A \quad \textit{implies} \quad \Delta \vdash M \mathcal{R} N : A.$$

(iv) **Condensing:** If $x \notin (\mathrm{FV}(\Delta) \cup \mathrm{FV}(M) \cup \mathrm{FV}(N) \cup \mathrm{FV}(B))$.

$$\Gamma, x{:}A, \Delta \vdash M \mathcal{R} N : B \quad \textit{implies} \quad \Gamma, \Delta \vdash M \mathcal{R} N : B$$

(v) **Spec:**

$$\Gamma, x{:}A, \Delta \vdash M_1 \mathcal{R} M_2 : B \quad \& \quad \Gamma \vdash N : A$$
$$\textit{implies}$$
$$\Gamma, \Delta\{x := N\} \vdash M_1\{x := N\} \mathcal{R} M_2\{x := N\} : B\{x := N\}.$$

(vi) **Precong:** If $\Gamma \vdash M, N : A$ and $\Gamma \vdash C[M], C[N] : B$ where M, N, $C[M]$ and $C[N]$ are objects, then

$$\Gamma \vdash M \mathcal{R} N : A \quad \textit{implies} \quad \Gamma \vdash C[M] \mathcal{R} C[N] : B.$$

(vii) **Comp:**

$$\Gamma \vdash M \widehat{\mathcal{R}} N : A \quad \textit{implies} \quad \Gamma \vdash M \mathcal{R} N : A.$$

(viii) **Sub-object:**

$$\text{If } \Gamma, x{:}A, \Delta \vdash M_1 \mathcal{R} N_1 : B \quad \text{and} \quad \Gamma \vdash M_2 \mathcal{R} N_2 : A \quad \text{and} \quad \Gamma \vdash A*$$
$$\text{and } \Gamma, \Delta\{x := M_2\} =_{ml} \Gamma, \Delta\{x := N_2\} \quad \text{and} \quad B\{x := M_2\} =_{ml} B\{x := N_2\}$$
$$\text{then } \Gamma, \Delta\{x := M_2\} \vdash M_1\{x := M_2\} \mathcal{R} N_1\{x := N_2\} : B\{x := M_2\}$$
$$\text{and } \Gamma, \Delta\{x := N_2\} \vdash M_1\{x := M_2\} \mathcal{R} N_1\{x := N_2\} : B\{x := N_2\}.$$

(ix) **Sub-type:**

$$\begin{aligned}
&\text{If } \Gamma, x{:}K, \Delta \vdash M \mathcal{R} N : B \quad \text{and} \quad \Gamma \vdash A, A' : K \quad \text{and} \quad \Gamma \vdash K\Box \\
&\text{and } \Gamma, \Delta\{x := A\} =_{ml} \Gamma, \Delta\{x := A'\} \quad \text{and} \quad B\{x := A\} =_{ml} B\{x := A'\} \\
&\text{then } \Gamma, \Delta\{x := A\} \vdash M_1\{x := A\} \mathcal{R} N_1\{x := A'\} : B\{x := A\} \\
&\text{and } \Gamma, \Delta\{x := A'\} \vdash M_1\{x := A\} \mathcal{R} N_1\{x := A'\} : B\{x := A'\}.
\end{aligned}$$

Definition 36

(i) The *compatible closure* of a confined relation $\mathcal{R}$ is the confined relation $\widehat{\mathcal{R}}$ defined by the rules in Figure 19.

(ii) A confined relation is *natural* iff the properties **Type Conversion**, **Context Conversion**, **Thinning**, and **Spec** hold for it.

(iii) A confined relation is a *precongruence* iff the rule **Precong** is valid. A *congruence* is a confined relation that is both a precongruence and an equivalence relation.

The definitions below provide a way to induce a confined relation from a ground relation and vice versa.

Definition 37

(i) Let Γ be a legal context. Then a Γ-*closure* is a substitution σ defined inductively on the size of Γ as follows:
- if $\Gamma \equiv []$ then σ is the identity substitution, and
- if $\Gamma \equiv x_1 : A_1, \ldots, x_n : A_n$, then σ is a simultaneous substitution $\{x_1 := M_1, \ldots, , x_n := M_n\}$ such that for every $i \in \{1, \ldots, n\}$,
$$\vdash M_i : A_i\{x_1 := M_1, \ldots, x_{i-1} := M_{i-1}\}$$

(ii) The *confined extension* of a ground relation $\mathcal{R}_G$ is the confined relation $\mathcal{R}$ such that $(\Gamma \vdash M \mathcal{R} N : A)$ iff for all Γ-closures σ, $M\sigma \mathcal{R}_G N\sigma$.

(iii) If $\mathcal{R}$ is a confined relation, then its *ground restriction* is the ground relation
$$\{(M, N) \mid \cdot \vdash M \mathcal{R} N : A\}$$

(iv) If $\mathcal{R}$ is a confined relation, write $M \mathcal{R} N$ to mean a pair (M, N) is in the ground restriction of $\mathcal{R}$.

The definition of closure is slightly more complicated than other presentations because in the MTS setting one must take dependencies in the context into account when defining the notion of closure. The following proposition establishes that the notion of closure that we have defined above is the correct one in the sense that it generalizes substitution of a single closed term.

Proposition 38 *Let* $\Gamma \vdash N : A$. *For all* Γ-*closures* σ, $\vdash N\sigma : A\sigma$.

Given a well-formed context Γ, it is possible that there exist no Γ-closures due to the fact that some types may be uninhabited. The following proposition characterizes when

closures may exist.

Proposition 39

- *If* $\Gamma \equiv [\cdot]$ *then a* Γ*-closure exists (namely, the identity substitution).*
- *If* $\Gamma \equiv \Delta, x : A$ *then a* Γ*-closure exists if there is a* Δ*-closure* σ *and a term* B *such that* $\vdash B : A\sigma$.

The following proposition states basic properties of confined relations.

Proposition 40

(i) The compatible closure of any reflexive confined relation is reflexive.

(ii) The confined extension of a ground relation is natural.

(iii) If $\mathcal{R}$ *is a ground relation, and if* Γ *is a context for which no closure exists, then for all* M,N *such that* $\Gamma \vdash M, N : A : *$, $\Gamma \vdash M \mathcal{R} N : A$.

Component (3) reflects the fact that uninhabited types may cause terms M,N to be trivially related by the confined extension of a ground relation. This in term, means that **Condensing** does not hold the confined extension of a ground relation. Referring to the definition of **Condensing** (see Definition 35), a $\Gamma, x : A, \Delta$-closure may not exist due to the type uninhabited A (and thus M and N are trivially related), but a Γ, Δ-closure may exist that causes M and N not to be related. This illustrates the importance of working with confined relations where the context is made explicit.

We can now define applicative similarity, the ground preorder from which we will define operational equality.

Definition 41 Given a ground relation R, let the ground relation $[R]$ between programs of the same type A be defined as follows:

$\cdot \vdash M\,[R]\,N : A$ iff whenever $M \Downarrow U$ there is a V with $N \Downarrow V$ and $\cdot \vdash U\,\widehat{R}\,V : A$.

An applicative simulation is a relation $\mathcal{S}$ such that $\mathcal{S} \subseteq [\mathcal{S}]$. Define *ground applicative similarity* $\leq_a$ to be union of all applicative simulations. Define *applicative similarity* $\leq_a$ to be the confined extension of ground applicative similarity.

Proposition 42

(i) Function $[\cdot]$ *is monotonic.*

(ii) The identity ground relation is an applicative simulation.

(iii) If $\mathcal{S}_1$ *and* $\mathcal{S}_2$ *are applicative simulations, then so is* $\mathcal{S}_1\mathcal{S}_2$.

(iv) Ground similarity is the greatest fixed-point of $[\cdot]$ *and is the greatest applicative simulation.*

(v) $\cdot \vdash M \leq_a N : A$ *iff there is an applicative simulation* $\mathcal{S}$ *such that* $\cdot \vdash M\,\mathcal{S}\,N : A$.

(vi) Ground applicative similarity is a preorder.

Definition 43 An *applicative bisimulation* is a relation $\mathcal{S}$ such that both $\mathcal{S}$ and $\mathcal{S}^{-1}$ are applicative simulations. *Ground applicative bisimilarity* $\simeq_a$ is the ground relation on

programs of type A such that

$$\cdot \vdash M \simeq_a N : A \quad \text{iff} \quad \cdot \vdash M \leq_a N : A \text{ and } \cdot \vdash N \leq_a M : A.$$

Applicative bisimilarity $\simeq_a$ is the confined extension of ground applicative bisimilarity. We take applicative similarity to be our notion of equality.

Since ground applicative similarity $\leq_a$ is a preorder (Proposition 42), its symmetric closure ground applicative bisimilarity a$\simeq_a$ is an equivalence relation. Furthermore, for any programs M and N of type A, $\cdot \vdash M \simeq_a N : A$ iff for some applicative bisimulation $\mathcal{S}$, $\cdot \vdash M \mathcal{S} N : A$.

Proposition 44 *Ground relations* $\leq_a$ *and* $\simeq_a$ *are both operationally adequate.*

We are now ready to prove that applicative similarity $\leq_a$ is a precongruence using a technique due to Howe [29]. The idea is to define another relation, compatible similarity $\leq_c$, that is obviously a precongruence and obviously contains $\leq_a$, and then show that $\leq_a$ contains $\leq_c$ (*i.e.*, the two relations are the same).

Definition 45 (compatible similarity) Define the confined relation *compatible similarity* $\leq_c$ to be the smallest relation closed under the following rule.

$$\frac{\Gamma \vdash L \widehat{\leq_c} M : A \qquad \Gamma \vdash M \leq_a N : A}{\Gamma \vdash L \leq_c N : A}$$

The following proposition gives a number of properties; the last is the desired property that $\leq_a$ is a precongruence.

Proposition 46

(i) Applicative similarity is natural.

(ii) Confined relations $\widehat{\leq_c}$ *and* $\leq_c$ *are reflexive.*

(iii) Confined relations $\widehat{\leq_c}$ *and* $\leq_a$ *both imply* $\leq_c$.

(iv) If $\Gamma \vdash M_1 \leq_c M_2 : A$ *and* $\Gamma \vdash M_2 \leq_a M_3 : A$ *then* $\Gamma \vdash M_1 \leq_c M_3 : A$.

(v) The rules **Sub-object** *and* **Sub-type** *(of Definition 35) hold for confined relation* $\leq_c$.

(vi) If $\cdot \vdash U \leq_c M : A$ *for value* U *then there is a value* V *such that* $M \Downarrow V$ *and* $\cdot \vdash U \widehat{\leq_c} V : A$.

(vii) If $\cdot \vdash M \leq_c N : A$ *and* $M \Downarrow U$ *then* $\cdot \vdash U \leq_c N : A$.

(viii) If $\Gamma \vdash M \leq_c N : A$ *then* $\Gamma \vdash M \leq_a N : A$.

(ix) Applicative similarity is a precongruence.

The fact that applicative similarity is a precongruence (and applicative bisimilarity is a congruence) allows us to reason compositionally about MTS programs. Given this, it is relatively straightforward to show that the equivalences of Figure 20 hold. As a

Congruence

$$\text{If } (\Gamma \vdash M \mathrel{\widehat{\simeq}_a} N : A) \text{ then } (\Gamma \vdash M \simeq_a N : A).$$

Canonical Exclusivity

$$\text{If } (\Gamma \vdash U \simeq_a V : A) \text{ then } (\Gamma \vdash U \mathrel{\widehat{\simeq}_a} V : A).$$

MTS Calculus

$$\Gamma \vdash (\lambda x{:}B.\ M)\ N \simeq_a M\{x := N\} : A$$

$$\Gamma \vdash \mathsf{fix}\ x{:}M.\ N \simeq_a N\{x := \mathsf{fix}\ x{:}M.\ N\} : A$$

$$\Gamma \vdash \mathsf{let}\ x{:}A \Leftarrow \mathsf{unit}\ N\ \mathsf{in}\ M \simeq_a M\{x := N\} : A$$

$$\Gamma \vdash \mathsf{let}\ x{:}A \Leftarrow M\ \mathsf{in}\ \mathsf{unit}\ x \simeq_a M : A$$

$$\Gamma \vdash \mathsf{let}\ x_2{:}A_2 \Leftarrow (\mathsf{let}\ x_1{:}A_1 \Leftarrow M_1\ \mathsf{in}\ M_2)\ \mathsf{in}\ M_3 \simeq_a \mathsf{let}\ x_1{:}A_1 \Leftarrow M_1\ \mathsf{in}\ (\mathsf{let}\ x_2{:}A_2 \Leftarrow M_2\ \mathsf{in}\ M_3) : A$$

$$\Gamma \vdash \mathsf{succ}\ \lceil n \rceil \simeq_a \lceil n+1 \rceil : A$$

$$\Gamma \vdash \mathsf{pred}\ \lceil n+1 \rceil \simeq_a \lceil n \rceil : A$$

$$\Gamma \vdash \mathsf{pred}\ \lceil 0 \rceil \simeq_a \lceil 0 \rceil : A$$

$$\Gamma \vdash \mathsf{if0}\ \lceil 0 \rceil\ N_1\ N_2 \simeq_a N_1 : A$$

$$\Gamma \vdash \mathsf{if0}\ \lceil n+1 \rceil\ N_1\ N_2 \simeq_a N_2 : A$$

Strictness Law

$$\Gamma \vdash \mathsf{let}\ x{:}B \Leftarrow \bot\ \mathsf{in}\ M \simeq_a \bot : A$$

Fig. 20. Laws of equality in MTS

consequence, we have that MTS calculus is sound with respect to operational equivalence. This result provides the semantic foundation for the MTS framework.

Theorem 47 (soundness of MTS calculus) *For any objects M and N such that $\Gamma \vdash M : A$ and $\Gamma \vdash N : A$, $M =_{ml} N$ implies $M \simeq_a N$.*

6 Assessment and related work

Monadic type systems may prove useful in several applications, some of which are summarized below. In addition, monadic type systems may find applications in logic but we have not explored this possibility thus far.

Intermediate language for partial evaluation:

The computational metalanguage is well-suited as an evaluation-order independent intermediate language for partial evaluation [27]. The monadic structure guarantees the soundness of partial evaluation in the presence of computational effects [32, 48], and it facilitates the extension of type specialization to a language with first-class references [14, 30]. These works either take place in an untyped setting [32], or in a simply typed setting [14, 27, 30].

We expect the MTS-framework to contribute to the definition of sound partial evaluation for polymorphically typed languages (where specialization with respect to types coexists with specialization wrt. values in a single framework) and possibly also to an extension of type specialization to encompass polymorphism. It would also be interesting to consider type-directed partial evaluation [11] in this framework.

Intermediate language for compilation:

Various researchers have advocated the use of the metalanguage as an evaluation-order independent intermediate language for compiling [7, 28]. Many features of the metalanguage such as the reliance on monads to encapsulate effects, and the use of pointed types to distinguish call-by-name from call-by-value and certainly-converging terms from possibly-diverging terms have been incorporated into a recent proposal for a simply-typed evaluation-order independent intermediate language [38].

We believe that the MTS framework and the results of this paper can help provide a technical foundation for combining ideas from the PTS-based intermediate language of Meijer and Peyton Jones [31] and the language of Peyton Jones *et al.* [38].

Generic CPS translation framework:

Hatcliff and Danvy [28] have developed a generic framework for reasoning about CPS translations of simply-typed languages by factorizing the translations through a simply-typed version of the computational metalanguage. The results presented in the present work enable us to generalize this idea to treat in a single framework call-by-value, call-by-name, and mixed strategy CPS translations for PTSs. The general translation could be instantiated to the translations for F_2 [23], F_ω [22], non-dependent PTSs [10], as well as the recent CPS translations for PTSs with dependent types proposed by Barthe *et al.* [6].

7 Conclusion and directions for further research

The paper introduces monadic type systems, a powerful and flexible framework to capture computational effects in rich (polymorphic, dependent) type systems, and study operational semantics for monadic type systems with dependent types.

Much work remains to be done. For example, it seems important to extend the type structure of monadic type systems with further constructs as the type structure of MTSs, with only two type constructors (monads and dependent products), is too minimal for some applications.

Section 5 presents an operational semantics for MTSs, which smoothly generalizes certain aspects of Gordon's earlier work [21]. A more extensive investigation needs to be done to compare applicative similarity to other familiar notions of equivalence such as contextual similarity. In addition, the semantics is mainly concerned with the lifting monads. It would be interesting to scale up alternative semantics for notions of computation (see *e.g.*, [43]) to the framework of MTSs.

Finally, we are interested in exploiting our framework for CPS translation and partial evaluation. Based on the technical results of [6], we are currently extending the results of [28] to MTSs.

References

[1] P. Audebaud. Partial objects in the calculus of constructions. In *Proceedings of LICS'91*, pages 86–95. IEEE Computer Society Press, 1991.

[2] H. Barendregt. *The Lambda Calculus: Its Syntax and Semantics*, volume 103 of *Studies in Logic and the Foundations of Mathematics*. North-Holland, revised edition, 1984.

[3] H. Barendregt. Introduction to Generalised Type Systems. *Journal of Functional Programming*, 1(2):125–154, April 1991.

[4] H. Barendregt. Lambda calculi with types. In S. Abramsky, D. Gabbay, and T. Maibaum, editors, *Handbook of Logic in Computer Science*, pages 117–309. Oxford Science Publications, 1992. Volume 2.

[5] G. Barthe. The relevance of proof-irrelevance. Manuscript, 1997.

[6] Gilles Barthe, John Hatcliff, and Morten Heine Sørensen. CPS translations and applications: the cube and beyond. In Olivier Danvy, editor, *Proceedings of the Second ACM SIGPLAN Workshop on Continuations*, number NS-96-13 in BRICS Notes, pages 4–1–4–31, Paris, France, January 1997. Dept. of Computer Science, Aarhus, Denmark.

[7] N. Benton. *Strictness Analysis of Lazy Functional Programs*. PhD thesis, University of Cambridge, 1992.

[8] L. Cardelli. Phase distinctions in type theory. Unpublished Mansucript, January 1988.

[9] T. Coquand. Typechecking dependent types. Privately circulated manuscript, 1998.

[10] T. Coquand and H. Herbelin. A-translation and looping combinators in pure type systems. *Journal of Functional Programming*, 4(1):77–88, January 1994.

[11] O. Danvy. Type-directed partial evaluation. In *Proceedings of POPL'96*, pages 242–257. ACM Press, 1996.

[12] Olivier Danvy and John Hatcliff. Cps transformation after strictness analysis. *Letters on Programming Languages and Systems*, 1(3):195–212, 1993.

[13] Olivier Danvy and John Hatcliff. On the transformation between direct and continuation semantics. In Stephen Brookes, Michael Main, Austin Melton, Michael Mislove, and David Schmidt, editors, *Proc. of the 9th Conference on Mathematical Foundations of Programming Semantics*, volume 802 of *Lecture Notes in Computer Science*, pages 627–638, New Orleans, Louisiana, April 1993. Springer-Verlag.

[14] D. Dussart, J. Hughes, and P. Thiemann. Type specialisation for imperative languages. In *Proceedings of ICFP'97*, pages 204–216. ACM Press, 1997.

[15] Matthias Felleisen and Robert Hieb. The revised report on the syntactic theories of sequential control and state. *Theoretical Computer Science*, 10(2):235–271, 1992.

[16] Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. The essence of compiling with continuations. In *Proc. of the ACM SIGPLAN '93 Conference on Programming Language Design and Implementation*, pages 237–247, Albuquerque, New Mexico, June 1993.

[17] J. Garrigue and D. Rémy. Extending ML with semi-explicit higher-order polymorphism. In *Proceedings of TACS'97*, volume 1281 of *Lecture Notes in Computer Science*, pages 20–46. Springer-Verlag, 1997.

[18] H. Geuvers. *Logics and type systems.* PhD thesis, University of Nijmegen, 1993.

[19] J-Y. Girard. *Interprétation fonctionelle et élimination des coupures dans l'arithmétique d'ordre supérieur.* Thèse d'Etat, Université Paris 7, 1972.

[20] J.-Y. Girard, Y. Lafont, and P. Taylor. *Proofs and Types.* Number 7 in Tracts in Theoretical Computer Science. Cambridge University Press, 1989.

[21] Andrew D. Gordon. *Functional Programming and Input/Output.* Cambridge University Press, 1994.

[22] R. Harper and M. Lillibridge. Explicit polymorphism and CPS conversion. In *Proceedings of POPL'93*, pages 206–219. ACM Press, 1993.

[23] Robert Harper and Mark Lillibridge. Polymorphic type assignment and cps conversion. *Lisp and Symbolic Computation*, 6(4):361–380, 1993.

[24] Robert Harper and Mark Lillibridge. Operational interpretations of an extension of f_ω with control operators. *Journal of Functional Programming*, 6(3):393–417, May 1996.

[25] Robert Harper and John C. Mitchell. On the type structure of standard ML. *ACM Transactions on Programming Languages and Systems*, 15(2):211–252, April 1993.

[26] Robert Harper and Greg Morrisett. Compiling polymorphism using intensional type analysis. In *Proc. 22nd Annual ACM Symposium on Principles of Programming Languages*, pages 130–141, San Francisco, CA, January 1995. ACM Press.

[27] J. Hatcliff and O. Danvy. A computational formalization for partial evaluation. *Mathematical Structures in Computer Science*, 7(5):507–541, October 1997.

[28] John Hatcliff and Olivier Danvy. A generic account of continuation-passing styles. In *Proc. 21st Annual ACM Symposium on Principles of Programming Languages*, pages 458–471, Portland, OG, January 1994. ACM Press.

[29] D.J. Howe. Proving congruence of bisimulation in functional programming languages. *Information and Computation*, 124(2):103–112, February 1996.

[30] John Hughes. Type specialisation for the λ-calculus; or, a new paradigm for partial evaluation based on type inference. In Olivier Danvy, Robert Glück, and Peter Thiemann, editors, *Partial Evaluation*, volume 1110 of *Lecture Notes in Computer Science*, pages 183–215, Dagstuhl, Germany, February 1996. Springer Verlag, Heidelberg.

[31] S. Peyton Jones and E. Meijer. Henk: a typed intermediate language. Proceedings of the ACM Workshop on Types in Compilation, 1997.

[32] J. Lawall and P. Thiemann. Sound specialization in the presence of computational effects. In *Proceedings of TACS'97*, volume 1281 of *Lecture Notes in Computer Science*, pages 165–190. Springer-Verlag, 1997.

[33] X. Leroy. Polymorphism by name for references and continuations. In *Proceedings of POPL'93*, pages 220–231. ACM Press, 1993.

[34] R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML (Revised)*. The MIT Press, 1997.

[35] Yasuhiko Minamide, Greg Morrisett, and Robert Harper. Typed closure conversion. In *Proc. 23rd Annual ACM Symposium on Principles of Programming Languages*, pages 271–283, St. Petersburg, Fla., January 1996. ACM Press.

[36] Eugenio Moggi. Notions of computations and monads. *Information and Computation*, 93:55–92, 1991.

[37] M. Odersky and K. Läufer. Putting type annotations to work. In *Proceedings of POPL'96*, pages 54–67. ACM Press, 1996.

[38] S. Peyton Jones, J. Launchbury, M. Shields, and A. Tolmach. The design of a typed intermediate language. In *Proceedings of POPL'98*. ACM Press, 1998.

[39] Simon L Peyton Jones, Cordelia Hall, Kevin Hammond, Will Partain, and Philip Wadler. The Glasgow Haskell compiler: a technical overview. In *Proceedings of the UK Joint Framework for Information Technology (JFIT) Technical Conference*, Keele, 1993.

[40] S.L. Peyton Jones and P.L. Wadler. Imperative functional programming. In *Proceedings of POPL'93*, pages 71–84. ACM Press, 1993.

[41] R. Pollack. *The Theory of LEGO: A Proof Checker for the Extended Calculus of Constructions*. PhD thesis, University of Edinburgh, 1994.

[42] John C. Reynolds. Definitional interpreters for higher-order programming languages. In *ACM Annual Conference*, pages 717–740, July 1972.

[43] E. Ritter and A.M. Pitts. A fully abstract translation between a λ-calculus with reference types and Standard ML. In M. Dezani-Ciancaglini and G. Plotkin, editors, *Proceedings of TLCA'95*, volume 902 of *Lecture Notes in Computer Science*, pages 397–413. Springer-Verlag, 1995.

[44] Amr Sabry and John Field. Reasoning about explicit and implicit representations of state. In Paul Hudak, editor, *SIPL '93, ACM SIGPLAN Workshop on State in Programming Languages*, pages 17–30, Copenhagen, Denmark, June 1993. Yale University, Department of Computer Science, New Haven, CT. Technical Report YALEU/DCS/RR-968.

[45] Zhong Shao. Typed common intermediate format. In *Conference on Domain-Specific Languages*, Santa Barbara, CA, October 1997. USENIX.

[46] Zhong Shao and Andrew W. Appel. A type-based compiler for Standard ML. In *Proc. of the ACM SIGPLAN '95 Conference on Programming Language Design and Implementation*, La Jolla, CA, USA, June 1995. ACM Press.

[47] Dave Tarditi, Greg Morrisett, P. Cheng, Chris Stone, Robert Harper, and Peter Lee. TIL: A type-directed optimizing compiler for ML. In *Proc. of the ACM SIGPLAN '96 Conference on Programming Language Design and Implementation*, pages 181–192, Philadelphia, PA, USA, May 1996. ACM Press.

[48] Peter Thiemann and Dirk Dussart. Partial evaluation for higher-order languages with state. Berichte des Wilhelm-Schickard-Instituts WSI-97-XX, Universitt Tbingen, April 1997.

[49] Mads Tofte. Type inference for polymorphic references. *Information and Computation*, 89:1–34, 1990.

[50] Philip L. Wadler. The essence of functional programming. In *Proc. 19th Annual ACM Symposium on Principles of Programming Languages*, pages 1–14, Albuquerque, New Mexico, January 1992. ACM Press.

Proofs

A Example Compile-time Transformation

Consider the ML source term $(\lambda x.f@(!\ x))@(\mathsf{ref}\ 5)$ where $f : \mathsf{nat} \to \tau$. This term is transformed by $\mathcal{C}^M$ to

$$
\begin{array}{ll}
 & \mathsf{let}\ x_1 : \mathsf{ref}\ \mathsf{nat} \to \mathsf{M}\ \tau \Leftarrow \mathcal{C}^M[\![\lambda x.f@(!\ x)]\!]\ \mathsf{in} \\
 & \mathsf{let}\ x_2 : \mathsf{ref}\ \mathsf{nat} \Leftarrow \mathcal{C}^M[\![\mathsf{ref}\ 5]\!]\ \mathsf{in}\ x_1\ x_2 \\
\equiv & \mathsf{let}\ x_1 : \mathsf{ref}\ \mathsf{nat} \to \mathsf{M}\ \tau \Leftarrow \mathsf{unit}\ \lambda x.\mathcal{C}^M[\![f@(!\ x)]\!]\ \mathsf{in} \\
 & \mathsf{let}\ x_2 : \mathsf{ref}\ \mathsf{nat} \Leftarrow (\mathsf{let}\ x_3 : \mathsf{nat} \Leftarrow \mathsf{unit}\ 5\ \mathsf{in}\ \mathsf{ref}_M\ \mathsf{nat}\ x_3)\ \mathsf{in}\ x_1\ x_2 \\
\equiv & \mathsf{let}\ x_1 : \mathsf{ref}\ \mathsf{nat} \to \mathsf{M}\ \tau \Leftarrow \mathsf{unit}\ \lambda x. \\
 & \quad \mathsf{let}\ x_4 : \mathsf{nat} \to \mathsf{M}\ \tau \Leftarrow \mathsf{unit}\ f\ \mathsf{in} \\
 & \quad \mathsf{let}\ x_5 : \mathsf{nat} \Leftarrow (\mathsf{let}\ x_6 : \mathsf{ref}\ \mathsf{nat} \Leftarrow \mathsf{unit}\ x\ \mathsf{in}\ !_M\ \mathsf{nat}\ x_6)\ \mathsf{in}\ x_4\ x_5\ \mathsf{in} \\
 & \mathsf{let}\ x_2 : \mathsf{ref}\ \mathsf{nat} \Leftarrow (\mathsf{let}\ x_3 : \mathsf{nat} \Leftarrow \mathsf{unit}\ 5\ \mathsf{in}\ \mathsf{ref}_M\ \mathsf{nat}\ x_3)\ \mathsf{in}\ x_1\ x_2 \\
\Rightarrow_{ml} & \mathsf{let}\ x_1 : \mathsf{ref}\ \mathsf{nat} \to \mathsf{M}\ \tau \Leftarrow \mathsf{unit}\ \lambda x. \\
 & \quad \mathsf{let}\ x_4 : \mathsf{nat} \to \mathsf{M}\ \tau \Leftarrow \mathsf{unit}\ f\ \mathsf{in} \\
 & \quad \mathsf{let}\ x_6 : \mathsf{ref}\ \mathsf{nat} \Leftarrow \mathsf{unit}\ x\ \mathsf{in} \\
 & \quad \mathsf{let}\ x_5 : \mathsf{nat} \Leftarrow !_M\ \mathsf{nat}\ x_6\ \mathsf{in}\ x_4\ x_5\ \mathsf{in} \\
 & \mathsf{let}\ x_3 : \mathsf{nat} \Leftarrow \mathsf{unit}\ 5\ \mathsf{in} \\
 & \mathsf{let}\ x_2 : \mathsf{ref}\ \mathsf{nat} \Leftarrow \mathsf{ref}_M\ \mathsf{nat}\ x_3\ \mathsf{in}\ x_1\ x_2 \\
\Rightarrow_{ml} & \mathsf{let}\ x_2 : \mathsf{ref}\ \mathsf{nat} \Leftarrow \mathsf{ref}_M\ \mathsf{nat}\ 5\ \mathsf{in} \\
 & (\lambda x.\mathsf{let}\ x_5 : \mathsf{nat} \Leftarrow !_M\ \mathsf{nat}\ x\ \mathsf{in}\ f\ x_5)\ x_2 \\
\Rightarrow_{ml} & \mathsf{let}\ x_2 : \mathsf{ref}\ \mathsf{nat} \Leftarrow \mathsf{ref}_M\ \mathsf{nat}\ 5\ \mathsf{in} \\
 & \mathsf{let}\ x_5 : \mathsf{nat} \Leftarrow !_M\ \mathsf{nat}\ x_2\ \mathsf{in}\ f\ x_5
\end{array}
$$

If the compiler added special rules for reasoning with state [44] it could derive an even more optimized program.

$$\mathsf{let}\ x_2 : \mathsf{ref}\ \mathsf{nat} \Leftarrow \mathsf{ref}_M\ \mathsf{nat}\ 5\ \mathsf{in}\ f\ 5$$

The pervasive type annotations in the program can guide the compiler to choose good (unboxed) runtime representations for 5 and for ref 5.

B Strong normalization of $\mu\kappa$-reduction

B.1 Finiteness of Developments

A development is a rewrite sequence in which the contracted redexes are descendants of the redexes of the original term. A standard result in λ-calculus, called Finiteness of Developments, shows that all such reduction sequences are finite. Among the many possible proofs of FD, one proceeds by defining underlined λ-terms. We take this path here.

Definition 48 The set $\mathcal{U}$ of *underlined λ-terms* is defined by the abstract syntax:

$$\mathcal{U} = V \mid \mathcal{U}\,\mathcal{U} \mid \lambda V.\,\mathcal{U} \mid (\underline{\lambda} V.\,\mathcal{U})\,\mathcal{U}$$

The notion $\underline{\beta}$ of *underlined β-reduction* is defined by the contraction rule

$$(\underline{\lambda} x.\,M)\,N \quad \to_{\underline{\beta}} \quad M\{x := N\}$$

Finiteness of Developments can be stated as:

Theorem 49 $\mathcal{U} = \mathrm{SN}(\underline{\beta})$.

B.2 Commuting conversions

In order to prove strong normalisation of μ-reduction, we need to consider a commuting conversion $\underline{\beta'}$.

Definition 50 The notion of reduction $\underline{\beta'}$ is defined by the contraction rule

$$P\,((\underline{\lambda} x.\,Q)\,R) \quad \to_{\underline{\beta'}} \quad (\underline{\lambda} x.\,P\,Q)\,R$$

provided $x \notin \mathrm{FV}(P)$.

$\underline{\beta'}$ preserves strong normalisation, in the sense that underlined λ-terms are $\underline{\beta\beta'}$-strongly normalising.

Theorem 51 $\mathcal{U} = \mathrm{SN}(\underline{\beta\beta'})$.

Proof Every term $M \in \mathcal{U}$ is $\underline{\beta}$-strongly normalising, so we can define $\mathsf{maxred}(M)$ to be the length of the longest $\underline{\beta}$-reduction sequence starting from M. Now assume that $M \notin \mathrm{SN}(\underline{\beta\beta'})$, so we have

$$M \to_{\underline{\beta\beta'}} M_1 \to_{\underline{\beta\beta'}} \cdots$$

Without loss of generality, we can assume that:

(i) $N \in \mathrm{SN}(\underline{\beta\beta'})$ for every $N \in \mathcal{U}$ such that $\mathsf{maxred}(N) < \mathsf{maxred}(M)$;

(ii) $N \in \mathrm{SN}(\underline{\beta\beta'})$ for every strict subterm N of M.

We now proceed by case analysis on M. Necessarily M is of the form

$$(\underline{\lambda} x.\,P)\,Q\,R_1\,\ldots\,R_n$$

Moreover, the first reduction step $M \to_{\underline{\beta\beta'}} M_1$ is a $\underline{\beta'}$-step, otherwise we would reach a contradiction. There are four possibilities, out of which we treat the last one in detail:

- $M \to_{\underline{\beta'}} (\underline{\lambda}x.\, P')\, Q\, R_1\, \ldots\, R_n$ with $P \to_{\underline{\beta'}} P'$.
- $M \to_{\underline{\beta'}} (\underline{\lambda}x.\, P)\, Q'\, R_1\, \ldots\, R_n$ with $Q \to_{\underline{\beta'}} Q'$.
- $M \to_{\underline{\beta'}} (\underline{\lambda}x.\, P)\, Q\, R_1\, \ldots\, R_i'\, \ldots R_n$ for some i with $R_i \to_{\underline{\beta'}} R_i'$.
- $Q \equiv (\underline{\lambda}y.\, S)\, T$ with $y \notin \mathrm{FV}(P)$ and $M \to_{\underline{\beta'}} (\underline{\lambda}y.\, (\underline{\lambda}x.\, P)\, S)\, T\, R_1\, \ldots\, R_n$. By 2, each subterm of M is $\underline{\beta\beta'}$-strongly normalising, so necessarily the reduction sequence must contract a descendant of one of the redexes $(\underline{\lambda}x.\, P)\, S$ or $(\underline{\lambda}y.\, (\underline{\lambda}x.\, P)\, S)\, T$. In other words, the reduction sequence will be of the form

$$\begin{aligned}(\underline{\lambda}y.\, (\underline{\lambda}x.\, P)\, S)\, T\, R_1\, \ldots\, R_n &\twoheadrightarrow_{\underline{\beta\beta'}} (\underline{\lambda}y.\, (\underline{\lambda}x.\, P')\, S')\, T'\, R_1'\, \ldots\, R_n' \\ &\to_{\underline{\beta}} (\underline{\lambda}x.\, P')\, (S'\{y := T'\})\, R_1'\, \ldots\, R_n' \\ &\to_{\underline{\beta}} \ldots\end{aligned}$$

  or

$$\begin{aligned}(\underline{\lambda}y.\, (\underline{\lambda}x.\, P)\, S)\, T\, R_1\, \ldots\, R_n &\twoheadrightarrow_{\underline{\beta\beta'}} (\underline{\lambda}y.\, (\underline{\lambda}x.\, P')\, S')\, T'\, R_1'\, \ldots\, R_n' \\ &\to_{\underline{\beta}} (\underline{\lambda}y.\, (P'\{x := S'\}))\, T'\, R_1'\, \ldots\, R_n' \\ &\to_{\underline{\beta}} \ldots\end{aligned}$$

  with $X \twoheadrightarrow_{\underline{\beta\beta'}} X'$ for every term X.
  - The first case is impossible since $M \to_{\underline{\beta}} (\underline{\lambda}x.\, P)\, (S\{y := T\})\, R_1\, \ldots\, R_n$ and hence by assumption, $(\underline{\lambda}x.\, P)\, (S\{y := T\})\, R_1\, \ldots\, R_n \in \mathrm{SN}(\underline{\beta\beta'})$. On the other hand, $(\underline{\lambda}x.\, P')\, (S'\{y := T'\})\, R_1'\, \ldots\, R_n' \notin \mathrm{SN}(\underline{\beta\beta'})$ and $(\underline{\lambda}x.\, P)\, (S\{y := T\})\, R_1\, \ldots\, R_n \twoheadrightarrow_{\underline{\beta\beta'}} (\underline{\lambda}x.\, P')\, (S'\{y := T'\})\, R_1'\, \ldots\, R_n'$, a contradiction.
  - The second case is impossible. Indeed, there are two possibilities:
    $P'\{x := S'\} \notin \mathrm{SN}(\underline{\beta\beta'})$. This possibility is to be ruled out since $P\{x := S\}\{y := T\} \in \mathrm{SN}(\underline{\beta\beta'})$ by 1 and thus $P\{x := S\} \in \mathrm{SN}(\underline{\beta\beta'})$, which together with $P\{x := S\} \twoheadrightarrow_{\underline{\beta\beta'}} P''\{x := S''\}$ contradicts $P''\{x := S''\} \notin \mathrm{SN}(\underline{\beta\beta'})$.
    the infinite reduction sequence proceeds further as

$$\begin{aligned}(\underline{\lambda}y.\, (P'\{x := S'\}))\, T'\, R_1'\, \ldots\, R_n' &\twoheadrightarrow_{\underline{\beta\beta'}} (\underline{\lambda}y.\, (P''\{x := S''\}))\, T''\, R_1''\, \ldots\, R_n'' \\ &\to_{\underline{\beta}} (P''\{x := S''\}\{y := T''\})\, R_1''\, \ldots\, R_n'' \\ &\to_{\underline{\beta}} \ldots\end{aligned}$$

    This possibility is to be ruled out since $M \twoheadrightarrow_{\underline{\beta}} (P\{x := S\}\{y := T\})\, R_1\, \ldots\, R_n$ so by 1, $(P\{x := S\}\{y := T\})\, R_1\, \ldots\, R_n \in \mathrm{SN}(\underline{\beta\beta'})$, which together with $(P\{x := S\}\{y := T\})\, R_1\, \ldots\, R_n \twoheadrightarrow_{\underline{\beta\beta'}} (P''\{x := S''\}\{y := T''\})\, R_1''\, \ldots\, R_n''$ contradicts $(P''\{x := S''\}\{y := T''\})\, R_1''\, \ldots\, R_n'' \notin \mathrm{SN}(\underline{\beta\beta'})$.

■

C Proof of Lemma 2

Let $\Gamma = \{y_1 : \sigma_1, \ldots, y_n : \sigma_n\}$. Let further $\Gamma' = \{y_1 : A_1', \ldots, y_n : A_n'\}$ where $A_i' = \mathcal{S}^M[\![A_i]\!]$ if y_i is not fix-bound and $A_i' = \mathcal{C}^M[\![A_i]\!]$ otherwise. Then

(C.1) $$\Gamma \vdash_{SML} e : \tau \quad \Rightarrow \quad G_\Gamma, \Gamma' \vdash \mathcal{C}^M[\![e]\!] : \mathcal{C}^M[\![\tau]\!]$$

using $\mathcal{R}_{ML}$.

Proof The proof is by simultaneous induction on the type derivation of an expression using the auxiliary inductive hypothesis:

$$\text{(C.2)} \qquad \Gamma \vdash_{SML} v : \tau \quad \Rightarrow \quad G_\Gamma, \Gamma' \vdash \mathcal{V}^M[\![v]\!] : \mathcal{V}^M[\![\tau]\!].$$

First of all, G_Γ, Γ' is valid since for all $y_i : \mathcal{S}^M[\![\sigma_i]\!]$ in Γ' $G_\Gamma \vdash \mathcal{S}^M[\![\sigma_i]\!] : s$ where $s \in \{*^v, *^c, \sharp\}$.

- Hypothesis (C.2) is immediate for $\Gamma \vdash_{SML} i : \mathsf{nat}$, since $G_\Gamma, \Gamma' \vdash i : \mathcal{V}^M[\![\mathsf{nat}]\!]$ by the presence of constants.
- Hypothesis (C.2) for $\Gamma, x : \sigma \vdash_{SML} x : \tau \quad \tau \leq \sigma$. Suppose $x : \sigma$ in Γ with $\sigma = \forall\alpha_1 \ldots \alpha_n.\tau'$ and $\tau = \tau'\{\alpha_i \mapsto \tau_i\}$. Then

  $$G_\Gamma, \Gamma' \vdash \mathsf{unit}\ (x\ \mathcal{V}^M[\![\tau_1]\!]\ \ldots\ \mathcal{V}^M[\![\tau_n]\!]) : \mathcal{C}^M[\![\tau]\!]$$

  must be shown. By definition of Γ',

  $$G_\Gamma, \Gamma' \vdash x : \mathcal{S}^M[\![\forall\alpha_1 \ldots \alpha_n.\tau']\!]$$

  which is the same as

  $$G_\Gamma, \Gamma' \vdash x : \Pi\alpha_1{:}*^v. \ldots \Pi\alpha_n{:}*^v. \mathcal{V}^M[\![\tau']\!].$$

  Hence

  $$G_\Gamma, \Gamma' \vdash x\ \mathcal{V}^M[\![\tau_1]\!]\ \ldots\ \mathcal{V}^M[\![\tau_n]\!] : \mathcal{V}^M[\![\tau'\{\alpha_i \mapsto \tau_i\}]\!].$$

- Hypothesis (C.2) for $\Gamma, f : \tau_2 \to \tau_1 \vdash_{SML} f : \tau_2 \to \tau_1$. Hence $f : \mathcal{C}^M[\![\tau_2 \to \tau_1]\!]$ in Γ' and we have to prove

  $$G_\Gamma, \Gamma' \vdash \lambda y_2 : \mathcal{V}^M[\![\tau_2]\!].\mathsf{let}\ y_1 : \mathcal{V}^M[\![\tau_2 \to \tau_1]\!] \Leftarrow f\ \mathsf{in}\ y_1\ y_2 : \mathcal{C}^M[\![\tau_2 \to \tau_1]\!].$$

  By the (application) rule

  $$\text{(C.3)} \qquad G_\Gamma, \Gamma', y_2 : \mathcal{V}^M[\![\tau_2]\!], y_1 : \mathcal{V}^M[\![\tau_2]\!] \to \mathcal{C}^M[\![\tau_1]\!] \vdash y_1\ y_2 : \mathcal{C}^M[\![\tau_1]\!]$$

  and by the assumption on f

  $$\text{(C.4)} \qquad G_\Gamma, \Gamma', y_2 : \mathcal{V}^M[\![\tau_2]\!] \vdash f : \mathcal{C}^M[\![\tau_2 \to \tau_1]\!].$$

  Applying (let) to (C.3) and (C.4) yields

  $$G_\Gamma, \Gamma', y_2 : \mathcal{V}^M[\![\tau_2]\!] \vdash \mathsf{let}\ y_1 : \mathcal{V}^M[\![\tau_2 \to \tau_1]\!] \Leftarrow f\ \mathsf{in}\ y_1\ y_2 : \mathcal{C}^M[\![\tau_1]\!]$$

  And the claim follows by (abstraction).

  $$G_\Gamma, \Gamma' \vdash \lambda y_2 : \mathcal{V}^M[\![\tau_2]\!].\mathsf{let}\ y_1 : \mathcal{V}^M[\![\tau_2 \to \tau_1]\!] \Leftarrow f\ \mathsf{in}\ y_1\ y_2 : \mathcal{V}^M[\![\tau_2 \to \tau_1]\!].$$

- Hypothesis (C.2) for $\dfrac{\Gamma, x : \tau_2 \vdash_{SML} e : \tau_1}{\Gamma \vdash_{SML} \lambda x.e : \tau_2 \to \tau_1}$. By induction and $\mathcal{S}^M[\![\tau_2]\!] = \mathcal{V}^M[\![\tau_2]\!]$,

  $$G_\Gamma, \Gamma', x : \mathcal{V}^M[\![\tau_2]\!] \vdash \mathcal{C}^M[\![e]\!] : \mathcal{C}^M[\![\tau_1]\!].$$

  By the (abstraction) rule

  $$G_\Gamma, \Gamma' \vdash \lambda x : \mathcal{V}^M[\![\tau_2]\!].\mathcal{C}^M[\![e]\!] : \mathcal{V}^M[\![\tau_2]\!] \to \mathcal{C}^M[\![\tau_1]\!].$$

  Since $\mathcal{V}^M[\![\tau_2]\!] \to \mathcal{C}^M[\![\tau_1]\!] = \mathcal{V}^M[\![\tau_2 \to \tau_1]\!]$,

  $$G_\Gamma, \Gamma' \vdash \lambda x : \mathcal{V}^M[\![\tau_2]\!].\mathcal{C}^M[\![e]\!] : \mathcal{V}^M[\![\tau_2 \to \tau_1]\!].$$

- Hypothesis (C.2) for $\dfrac{\Gamma, f : \tau_2 \to \tau_1 \vdash_{SML} e : \tau_2 \to \tau_1}{\Gamma \vdash_{SML} \mathsf{fix}\ f.e : \tau_2 \to \tau_1}$. By induction and $\mathcal{C}^M[\![\tau]\!] = \mathsf{M}\ \mathcal{V}^M[\![\tau]\!]$,

$$G_\Gamma, \Gamma', f : \mathsf{M}\ \mathcal{V}^M[\![\tau_2 \to \tau_1]\!] \vdash \mathcal{C}^M[\![e]\!] : \mathsf{M}\ \mathcal{V}^M[\![\tau_2 \to \tau_1]\!]$$

By the (fixpoint) rule

$$G_\Gamma, \Gamma' \vdash \mathcal{C}^M[\![\mathsf{fix}\ f.\mathcal{C}^M[\![e]\!]]\!] : \mathsf{M}\ \mathcal{V}^M[\![\tau_2 \to \tau_1]\!]$$

and by (weakening)

(C.5) $$G_\Gamma, \Gamma', y_2 : \mathcal{V}^M[\![\tau_2]\!] \vdash \mathsf{fix}\ f.\mathcal{C}^M[\![e]\!] : \mathsf{M}\ \mathcal{V}^M[\![\tau_2 \to \tau_1]\!].$$

As seen before,

(C.6) $$G_\Gamma, \Gamma', y_2 : \mathcal{V}^M[\![\tau_2]\!], y_1 : \mathcal{V}^M[\![\tau_2]\!] \to \mathcal{C}^M[\![\tau_1]\!] \vdash y_1\ y_2 : \mathcal{C}^M[\![\tau_1]\!]$$

Apply (let) to (C.5) and (C.6) to obtain

(C.7) $G_\Gamma, \Gamma', y_2 : \mathcal{V}^M[\![\tau_2]\!] \vdash \mathsf{let}\ y_1 : \mathcal{V}^M[\![\tau_2]\!] \to \mathcal{C}^M[\![\tau_1]\!] \Leftarrow \mathsf{fix}\ f.\mathcal{C}^M[\![e]\!]\ \mathsf{in}\ y_1\ y_2 : \mathcal{C}^M[\![\tau_1]\!]$

Apply (abstraction) to (C.7)

$$G_\Gamma, \Gamma', y_2 : \mathcal{V}^M[\![\tau_2]\!] \vdash$$

$$\lambda y_2 : \mathcal{V}^M[\![\tau_2]\!].\mathsf{let}\ y_1 : \mathcal{V}^M[\![\tau_2]\!] \to \mathcal{C}^M[\![\tau_1]\!] \Leftarrow \mathsf{fix}\ f.\mathcal{C}^M[\![e]\!]\ \mathsf{in}\ y_1\ y_2 : \mathcal{V}^M[\![\tau_2]\!] \to \mathcal{C}^M[\![\tau_1]\!]$$

- Hypothesis (C.1) for values $\Gamma \vdash_{SML} v : \tau$. By the auxiliary inductive hypothesis (C.1),

$$G_\Gamma, \Gamma' \vdash \mathcal{V}^M[\![v]\!] : \mathcal{V}^M[\![\tau]\!]$$

By the (unit) rule

$$G_\Gamma, \Gamma' \vdash \mathsf{unit}\ (\mathcal{V}^M[\![v]\!]) : \mathsf{M}\ \mathcal{V}^M[\![\tau]\!]$$

and by the definition of $\mathcal{C}^M$

$$G_\Gamma, \Gamma' \vdash \mathcal{C}^M[\![v]\!] : \mathcal{C}^M[\![\tau]\!].$$

- $\dfrac{\Gamma \vdash_{SML} e_1 : \tau_2 \to \tau_1 \quad \Gamma \vdash_{SML} e_2 : \tau_2}{\Gamma \vdash_{SML} e_1 @ e_2 : \tau_1}$. By induction,

$$G_\Gamma, \Gamma' \vdash \mathcal{C}^M[\![e_1]\!] : \mathcal{C}^M[\![\tau_2 \to \tau_1]\!] \quad \text{and} \quad G_\Gamma, \Gamma' \vdash \mathcal{C}^M[\![e_2]\!] : \mathcal{C}^M[\![\tau_2]\!].$$

By rule (application)

(C.8) $$G_\Gamma, \Gamma', y_1 : \mathcal{V}^M[\![\tau_2]\!] \to \mathcal{C}^M[\![\tau_1]\!], y_2 : \mathcal{V}^M[\![\tau_2]\!] \vdash y_1\ y_2 : \mathcal{C}^M[\![\tau_2]\!].$$

By rule (let) applied to the inductive hypothesis for e_2 and (C.8)

(C.9) $G_\Gamma, \Gamma', y_1 : \mathcal{V}^M[\![\tau_2]\!] \to \mathcal{C}^M[\![\tau_1]\!] \vdash \mathsf{let}\ y_2 : \mathcal{V}^M[\![\tau_2]\!] \Leftarrow \mathcal{C}^M[\![e_2]\!]\ \mathsf{in}\ y_1\ y_2 : \mathcal{C}^M[\![\tau_2]\!]$

and again by rule (let) applied to the inductive hypothesis for e_1 and (C.9)

$$G_\Gamma, \Gamma' \vdash \mathsf{let}\ y_1 : \mathcal{V}^M[\![\tau_2]\!] \to \mathcal{C}^M[\![\tau_1]\!] \Leftarrow \mathcal{C}^M[\![e_1]\!]\ \mathsf{in}$$

$$\mathsf{let}\ y_2 : \mathcal{V}^M[\![\tau_2]\!] \Leftarrow \mathcal{C}^M[\![e_2]\!]\ \mathsf{in}\ y_1\ y_2 : \mathcal{C}^M[\![\tau_2]\!]$$

since $\mathcal{C}^M[\![\tau_2 \to \tau_1]\!] = \mathsf{M}\ (\mathcal{V}^M[\![\tau_2]\!] \to \mathcal{C}^M[\![\tau_1]\!])$.

- $\dfrac{\Gamma \vdash_{SML} v : \tau_1 \quad \Gamma, x : \mathsf{gen}(\Gamma, \tau_1) \vdash_{SML} e : \tau_2}{\Gamma \vdash_{SML} \mathsf{let}\ x = v\ \mathsf{in}\ e : \tau_2}$.

Call $\sigma = \mathsf{gen}(\Gamma, \tau_1) = \forall\alpha_1 \ldots \forall\alpha_n.\tau_1$. By induction (C.2) for v

(C.10) $$G_\Gamma, \Gamma' \vdash \mathcal{V}^M[\![v]\!] : \mathcal{V}^M[\![\tau_1]\!]$$

By (weakening)

(C.11) $$G_\Gamma, \Gamma', \alpha_1 : *^v, \ldots, \alpha_n : *^v \vdash \mathcal{V}^M[\![v]\!] : \mathcal{V}^M[\![\tau_1]\!]$$

By induction on n find

(C.12) $$G_\Gamma, \Gamma' \vdash \lambda\alpha_1 : *^v \ldots . \lambda\alpha_n : *^v . \mathcal{V}^M[\![v]\!] : \mathcal{S}^M[\![\sigma]\!]$$

By Lemma 1 there is an $s \in \{*^v, \sharp\}$ such that

(C.13) $$G_\sigma \vdash \mathcal{S}^M[\![\sigma]\!] : s$$

By definition of $\mathsf{gen}(\Gamma, \tau_1)$, G_σ is included in G_Γ, hence

(C.14) $$G_\Gamma \vdash \mathcal{S}^M[\![\sigma]\!] : s$$

By induction (C.1) on e and using (C.14) for the well-formedness of the assumptions

(C.15) $$G_\Gamma, \Gamma', x : \mathcal{S}^M[\![\sigma]\!] \vdash \mathcal{C}^M[\![e]\!] : \mathcal{C}^M[\![\tau_2]\!]$$

Applying (abstraction) to (C.15) yields

(C.16) $$G_\Gamma, \Gamma' \vdash \lambda x : \mathcal{S}^M[\![\sigma]\!] . \mathcal{C}^M[\![e]\!] : \mathcal{S}^M[\![\sigma]\!] \to \mathcal{C}^M[\![\tau_2]\!]$$

Apply (application) to (C.12) and (C.16) to obtain

(C.17) $$G_\Gamma, \Gamma' \vdash (\lambda x : \mathcal{S}^M[\![\sigma]\!] . \mathcal{C}^M[\![e]\!])\ (\lambda\alpha_1 : *^v \ldots . \lambda\alpha_n : *^v . \mathcal{V}^M[\![v]\!]) : \mathcal{C}^M[\![\tau_2]\!]$$

- $\dfrac{\Gamma \vdash_{SML} e_1 : \mathsf{nat} \quad \Gamma \vdash_{SML} e_2 : \tau \quad \Gamma \vdash_{SML} e_3 : \tau}{\Gamma \vdash_{SML} \mathsf{if0}\ e_1\ e_2\ e_3 : \tau}$. By induction

(C.18) $$G_\Gamma, \Gamma' \vdash \mathcal{C}^M[\![e_1]\!] : \mathcal{C}^M[\![\mathsf{nat}]\!]$$

and for $i = 2, 3$

(C.19) $$G_\Gamma, \Gamma' \vdash \mathcal{C}^M[\![e_i]\!] : \mathcal{C}^M[\![\tau]\!]$$

and by (weakening) and the definition of $\mathcal{C}^M$

(C.20) $$G_\Gamma, \Gamma', y : \mathsf{nat} \vdash \mathcal{C}^M[\![e_i]\!] : \mathsf{M}\ \mathcal{V}^M[\![\tau]\!].$$

By rule (if)

(C.21) $$G_\Gamma, \Gamma', y : \mathsf{nat} \vdash \mathsf{if0}\ y\ \mathcal{C}^M[\![e_2]\!]\ \mathcal{C}^M[\![e_3]\!] : \mathsf{M}\ \mathcal{V}^M[\![\tau]\!]$$

Applying (let) to (C.18) and (C.21) yields

(C.22) $$G_\Gamma, \Gamma' \vdash \mathsf{let}\ y : \mathsf{nat} \Leftarrow \mathcal{C}^M[\![e_1]\!]\ \mathsf{in}\ \mathsf{if0}\ y\ \mathcal{C}^M[\![e_2]\!]\ \mathcal{C}^M[\![e_3]\!] : \mathcal{C}^M[\![\tau]\!]$$

- $\dfrac{\Gamma \vdash_{SML} e : \mathsf{nat}}{\Gamma \vdash_{SML} \mathsf{succ}\ e : \mathsf{nat}}$. By induction,

(C.23) $$G_\Gamma, \Gamma' \vdash \mathcal{C}^M[\![e]\!] : \mathcal{C}^M[\![\mathsf{nat}]\!]$$

Starting from $y : \mathsf{nat}$, apply rule (succ)

(C.24) $$G_\Gamma, \Gamma', y : \mathsf{nat} \vdash \mathsf{succ}\ y : \mathsf{nat}$$

and then rule (unit)

(C.25) $$G_\Gamma, \Gamma', y : \mathsf{nat} \vdash \mathsf{unit}\ (\mathsf{succ}\ y) : \mathsf{M}\ \mathsf{nat}$$

Now recall $\mathsf{nat} \equiv \mathcal{V}^M[\![\mathsf{nat}]\!]$ and hence $\mathsf{M}\ \mathsf{nat} \equiv \mathcal{C}^M[\![\mathsf{nat}]\!]$ and apply (let) to (C.23) and (C.25) to obtain

(C.26) $$G_\Gamma, \Gamma' \vdash \mathsf{let}\ y : \mathsf{nat} \Leftarrow \mathcal{C}^M[\![e]\!]\ \mathsf{in}\ \mathsf{unit}\ (\mathsf{succ}\ y) : \mathcal{C}^M[\![\mathsf{nat}]\!]$$

- The case pred e is analogous.

■

D Proof of Theorem 5

Suppose $e_1 \to e_2$ in ML. Then $\mathcal{C}^M[\![e_1]\!] =_{ml} \mathcal{C}^M[\![e_2]\!]$.

Proof

- case $e_1 \equiv (\lambda x.e)@v$ and $e_2 \equiv e\{x := v\}$:

$$\begin{aligned}
& \mathcal{C}^M[\![e_1]\!] \\
\equiv\ & \mathsf{let}\ y_1 : \mathcal{V}^M[\![\tau_2 \to \tau_1]\!] \Leftarrow \mathcal{C}^M[\![\lambda x.e]\!]\ \mathsf{in}\ \mathsf{let}\ y_2 : \mathcal{V}^M[\![\tau_2]\!] \Leftarrow \mathcal{C}^M[\![v]\!]\ \mathsf{in}\ y_1\ y_2 \\
\equiv\ & \mathsf{let}\ y_1 : \mathcal{V}^M[\![\tau_2 \to \tau_1]\!] \Leftarrow \mathsf{unit}\ \lambda x : \mathcal{V}^M[\![\tau_2]\!].\mathcal{C}^M[\![e]\!]\ \mathsf{in} \\
& \mathsf{let}\ y_2 : \mathcal{V}^M[\![\tau_2]\!] \Leftarrow \mathsf{unit}\ \mathcal{V}^M[\![v]\!]\ \mathsf{in}\ y_1\ y_2 \\
\mapsto_{ml}\ & \mathsf{let}\ y_2 : \mathcal{V}^M[\![\tau_2]\!] \Leftarrow \mathsf{unit}\ \mathcal{V}^M[\![v]\!]\ \mathsf{in}\ \lambda x : \mathcal{V}^M[\![\tau_2]\!].\mathcal{C}^M[\![e]\!]\ y_2 \\
\mapsto_{ml}\ & \lambda x : \mathcal{V}^M[\![\tau_2]\!].\mathcal{C}^M[\![e]\!]\ \mathcal{V}^M[\![v]\!] \\
\mapsto_{ml}\ & \mathcal{C}^M[\![e]\!]\{x := \mathcal{V}^M[\![v]\!]\}
\end{aligned}$$

For e_2, $\mathcal{C}^M[\![e_2]\!] \equiv \mathcal{C}^M[\![e\{x := v\}]\!]$ which is $\equiv$ to $\mathcal{C}^M[\![e]\!]\{x := \mathcal{V}^M[\![v]\!]\}$ by Lemma 23 since $x : \mathcal{V}^M[\![\tau]\!]$.

- case $e_1 \equiv (\mathsf{fix}\ f.e)@v$ and $e_2 \equiv e\{f := \mathsf{fix}\ f.e\}@v$:

$$\begin{aligned}
& \mathcal{C}^M[\![e_1]\!] \\
\equiv\ & \mathsf{let}\ y_1 : \mathcal{V}^M[\![\tau_2 \to \tau_1]\!] \Leftarrow \mathcal{C}^M[\![\mathsf{fix}\ f.e]\!]\ \mathsf{in}\ \mathsf{let}\ y_2 : \mathcal{V}^M[\![\tau_2]\!] \Leftarrow \mathcal{C}^M[\![v]\!]\ \mathsf{in}\ y_1\ y_2 \\
\equiv\ & \mathsf{let}\ y_1 : \mathcal{V}^M[\![\tau_2 \to \tau_1]\!] \Leftarrow \mathsf{unit}\ (\lambda y_2 : \mathcal{V}^M[\![\tau_2]\!]. \\
& \qquad \mathsf{let}\ y_1 : \mathcal{V}^M[\![\tau_2 \to \tau_1]\!] \Leftarrow \mathsf{fix}\ f : \mathcal{C}^M[\![\tau_2 \to \tau_1]\!].\mathcal{C}^M[\![e]\!]\ \mathsf{in}\ y_1\ y_2)\ \mathsf{in} \\
& \mathsf{let}\ y_2 : \mathcal{V}^M[\![\tau_2]\!] \Leftarrow \mathsf{unit}\ \mathcal{V}^M[\![v]\!]\ \mathsf{in}\ y_1\ y_2 \\
\mapsto_{ml}\ & \mathsf{let}\ y_2 : \mathcal{V}^M[\![\tau_2]\!] \Leftarrow \mathsf{unit}\ \mathcal{V}^M[\![v]\!]\ \mathsf{in}\ (\lambda y_2 : \mathcal{V}^M[\![\tau_2]\!]. \\
& \qquad \mathsf{let}\ y_1 : \mathcal{V}^M[\![\tau_2 \to \tau_1]\!] \Leftarrow \mathsf{fix}\ f : \mathcal{C}^M[\![\tau_2 \to \tau_1]\!].\mathcal{C}^M[\![e]\!]\ \mathsf{in}\ y_1\ y_2)\ y_2 \\
\mapsto_{ml}\ & (\lambda y_2 : \mathcal{V}^M[\![\tau_2]\!].\mathsf{let}\ y_1 : \mathcal{V}^M[\![\tau_2 \to \tau_1]\!] \Leftarrow \mathsf{fix}\ f : \mathcal{C}^M[\![\tau_2 \to \tau_1]\!].\mathcal{C}^M[\![e]\!]\ \mathsf{in}\ y_1\ y_2)\ (\mathcal{V}^M[\![v]\!]) \\
\mapsto_{ml}\ & \mathsf{let}\ y_1 : \mathcal{V}^M[\![\tau_2 \to \tau_1]\!] \Leftarrow \mathsf{fix}\ f : \mathcal{C}^M[\![\tau_2 \to \tau_1]\!].\mathcal{C}^M[\![e]\!]\ \mathsf{in}\ y_1\ (\mathcal{V}^M[\![v]\!]) \\
\mapsto_{ml}\ & \mathsf{let}\ y_1 : \mathcal{V}^M[\![\tau_2 \to \tau_1]\!] \Leftarrow \mathcal{C}^M[\![e]\!]\{f := \mathsf{fix}\ f : \mathcal{C}^M[\![\tau_1 \to \tau_1]\!].\mathcal{C}^M[\![e]\!]\}\ \mathsf{in}\ y_1\ (\mathcal{V}^M[\![v]\!])
\end{aligned}$$

$$
\begin{aligned}
& \mathcal{C}^M[\![e_2]\!] \\
\equiv\ & \mathsf{let}\ y_1 : \mathcal{V}^M[\![\tau_2 \to \tau_1]\!] \Leftarrow \mathcal{C}^M[\![e\{f := \mathsf{fix}\ f.e\}]\!]\ \mathsf{in}\ \mathsf{let}\ y_2 : \mathcal{V}^M[\![\tau_2]\!] \Leftarrow \mathcal{C}^M[\![v]\!]\ \mathsf{in}\ y_1\ y_2 \\
=_{ml}\ & \mathsf{let}\ y_1 : \mathcal{V}^M[\![\tau_2 \to \tau_1]\!] \Leftarrow \mathcal{C}^M[\![e\{f := \mathsf{fix}\ f.e\}]\!]\ \mathsf{in}\ y_1\ (\mathcal{V}^M[\![v]\!])
\end{aligned}
$$

Here, we appeal to Lemma 22 to obtain

$$\mathcal{C}^M[\![e]\!]\{f := \mathsf{fix}\ f : \mathcal{C}^M[\![\tau_2 \to \tau_1]\!].\mathcal{C}^M[\![e]\!]\} \equiv \mathcal{C}^M[\![e\{f := \mathsf{fix}\ f.e\}]\!]$$

- case $e_1 \equiv \mathsf{let}\ x = v\ \mathsf{in}\ e$ and $e_2 \equiv e\{x := v\}$:

$$
\begin{aligned}
& \mathcal{C}^M[\![e_1]\!] \\
\equiv\ & (\lambda x : \mathcal{S}^M[\![\sigma]\!].\mathcal{C}^M[\![e]\!])\ (\lambda\alpha_1 : *^v \ldots \alpha_n : *^v.\mathcal{V}^M[\![v]\!]) \\
\mapsto_{ml}\ & \mathcal{C}^M[\![e]\!]\{x := (\lambda\alpha_1 : *^v \ldots \alpha_n : *^v.\mathcal{V}^M[\![v]\!])\}
\end{aligned}
$$

Convertibility $=_{ml}$ to $\mathcal{V}^M[\![e_2]\!]$ follows from Lemma 24.

- case $e_1 \equiv \mathsf{if0}\ 0\ e_1'\ e_2'$ and $e_2 \equiv e_1'$:

$$
\begin{aligned}
& \mathcal{C}^M[\![e_1]\!] \\
\equiv\ & \mathsf{let}\ y : \mathsf{nat} \Leftarrow \mathsf{unit}\ \lceil 0 \rceil\ \mathsf{in}\ \mathsf{if0}\ y\ \mathcal{C}^M[\![e_1']\!]\ \mathcal{C}^M[\![e_2']\!] \\
\mapsto_{ml}\ & \mathsf{if0}\ \lceil 0 \rceil\ \mathcal{C}^M[\![e_1']\!]\ \mathcal{C}^M[\![e_2']\!] \\
\mapsto_{ml}\ & \mathcal{C}^M[\![e_1']\!]
\end{aligned}
$$

- case $e_1 \equiv \mathsf{if0}\ (i+1)\ e_1'\ e_2'$ and $e_2 \equiv e_2'$: analogous.

- case $e_1 \equiv \mathsf{succ}\ i$ and $e_2 \equiv i + 1$:

$$
\begin{aligned}
& \mathcal{C}^M[\![e_1]\!] \\
\equiv\ & \mathsf{let}\ y : \mathsf{nat} \Leftarrow \mathsf{unit}\ \lceil i \rceil\ \mathsf{in}\ \mathsf{unit}\ (\mathsf{succ}\ y) \\
\mapsto_{ml}\ & \mathsf{unit}\ (\mathsf{succ}\ \lceil i \rceil) \\
=_{ml}\ & \mathsf{unit}\ \lceil i + 1 \rceil
\end{aligned}
$$

- case $e_1 \equiv \mathsf{pred}\ i$ and $e_2 \equiv i - 1$: analogous.

■

Lemma 22 *Suppose* $\Gamma, f : \tau_2 \to \tau_1 \vdash_{SML} e' : \tau'$ *and* $\Gamma \vdash_{SML} \mathsf{fix}\ f.e : \tau_2 \to \tau_1$ *then* $\mathcal{C}^M[\![e'\{f := \mathsf{fix}\ f.e\}]\!] \equiv \mathcal{C}^M[\![e']\!]\{f := \mathsf{fix}\ f : \mathcal{C}^M[\![\tau_2 \to \tau_1]\!].\mathcal{C}^M[\![e]\!]\}$.

Proof By induction on e, the only interesting case is $e' \equiv f$:

$$
\begin{aligned}
& \mathcal{C}^M[\![f\{f := \mathsf{fix}\ f.e\}]\!] \\
\equiv\ & \mathcal{C}^M[\![\mathsf{fix}\ f.e]\!] \\
\equiv\ & \mathsf{unit}\ (\lambda y_2 : \mathcal{V}^M[\![\tau_2]\!].\mathsf{let}\ y_1 : \mathcal{V}^M[\![\tau_2 \to \tau_1]\!] \Leftarrow \mathsf{fix}\ f : \mathcal{C}^M[\![\tau_2 \to \tau_1]\!].\mathcal{C}^M[\![e]\!]\ \mathsf{in}\ y_1\ y_2)
\end{aligned}
$$

$$
\begin{aligned}
& \mathcal{C}^M[\![f]\!]\{f := \mathsf{fix}\ f : \mathcal{C}^M[\![\tau_2 \to \tau_1]\!].\mathcal{C}^M[\![e]\!]\} \\
\equiv\ & \mathsf{unit}\ (\lambda y_2 : \mathcal{V}^M[\![\tau_2]\!].\mathsf{let}\ y_1 : \mathcal{V}^M[\![\tau_2 \to \tau_1]\!] \Leftarrow f\ \mathsf{in}\ y_1\ y_2)\{f := \mathsf{fix}\ f : \mathcal{C}^M[\![\tau_2 \to \tau_1]\!].\mathcal{C}^M[\![e]\!]\} \\
\equiv\ & \mathsf{unit}\ (\lambda y_2 : \mathcal{V}^M[\![\tau_2]\!].\mathsf{let}\ y_1 : \mathcal{V}^M[\![\tau_2 \to \tau_1]\!] \Leftarrow \mathsf{fix}\ f : \mathcal{C}^M[\![\tau_2 \to \tau_1]\!].\mathcal{C}^M[\![e]\!]\ \mathsf{in}\ y_1\ y_2)
\end{aligned}
$$

■

Lemma 23 *Suppose* $\Gamma, x : \tau \vdash_{SML} e : \tau'$ *and* $\Gamma \vdash_{SML} v : \tau$ *then* $\mathcal{C}^M[\![e\{x := v\}]\!] \equiv \mathcal{C}^M[\![e]\!]\{x := \mathcal{V}^M[\![v]\!]\}$.

Proof Induction on the proof of $\Gamma, x : \tau \vdash_{SML} e : \tau'$. ■

Lemma 24 *Suppose* $\Gamma, x : \forall\alpha_1 \ldots \forall\alpha_n.\tau \vdash_{SML} e : \tau'$, $\Gamma \vdash_{SML} v : \tau$, *and* $\forall\alpha_1 \ldots \forall\alpha_n.\tau = \mathsf{gen}(\Gamma, \tau)$ *then* $\mathcal{C}^M[\![e\{x := v\}]\!] =_{ml} \mathcal{C}^M[\![e]\!]\{x := \lambda\alpha_1 : *^v \ldots \alpha_n : *^v.\mathcal{V}^M[\![v]\!]\}$.

Proof By induction on e, the only interesting case is $e \equiv x$, $\tau' = \tau\{\alpha_i := \tau_i\}$:

$\mathcal{C}^M[\![x\{x := v\}]\!] \equiv \mathcal{C}^M[\![v]\!]\{\alpha_i := \mathcal{V}^M[\![\tau_i]\!]\}$

$$
\begin{aligned}
& \mathcal{C}^M[\![x]\!]\{x := \lambda\alpha_1 : *^v \ldots \alpha_n : *^v.\mathcal{V}^M[\![v]\!]\} \\
\equiv\ & \mathsf{unit}\ (x\ \mathcal{V}^M[\![\tau_1]\!]\ \ldots\ \mathcal{V}^M[\![\tau_n]\!])\{x := \lambda\alpha_1 : *^v \ldots \alpha_n : *^v.\mathcal{V}^M[\![v]\!]\} \\
\equiv\ & \mathsf{unit}\ ((\lambda\alpha_1 : *^v \ldots \alpha_n : *^v.\mathcal{V}^M[\![v]\!])\ \mathcal{V}^M[\![\tau_1]\!]\ \ldots\ \mathcal{V}^M[\![\tau_n]\!]) \\
=_{ml}\ & \mathsf{unit}\ (\mathcal{V}^M[\![v]\!]\{\alpha_i := \mathcal{V}^M[\![\tau_i]\!]\})
\end{aligned}
$$

■

E Proof of Lemma 4

Proof

- $\dfrac{\Gamma \vdash_{SML} e : \tau}{\Gamma \vdash_{SML} \mathsf{ref}\ e : \mathsf{ref}\ \tau}$. By induction,

(E.1) $$G_\Gamma, \Gamma' \vdash \mathcal{C}^M[\![e]\!] : \mathcal{C}^M[\![\tau]\!]$$

By Lemma 1 and rule (application),

(E.2) $$G_\Gamma, \Gamma' \vdash \mathsf{ref}_M\ \mathcal{V}^M[\![\tau]\!] : \mathcal{V}^M[\![\tau]\!] \to \mathsf{M}\ (\mathsf{ref}\ \mathcal{V}^M[\![\tau]\!])$$

By rule (weakening)

(E.3) $$G_\Gamma, \Gamma', y : \mathcal{V}^M[\![\tau]\!] \vdash \mathsf{ref}_M\ \mathcal{V}^M[\![\tau]\!] : \mathcal{V}^M[\![\tau]\!] \to \mathsf{M}\ (\mathsf{ref}\ \mathcal{V}^M[\![\tau]\!])$$

By rule (application) applied to (E.3) and $y : \mathcal{V}^M[\![\tau]\!]$,

(E.4) $$G_\Gamma, \Gamma', y : \mathcal{V}^M[\![\tau]\!] \vdash (\mathsf{ref}_M\ \mathcal{V}^M[\![\tau]\!])\ y : \mathsf{M}\ (\mathsf{ref}\ \mathcal{V}^M[\![\tau]\!])$$

Finally, apply rule (let) to (E.1) and (E.4)

(E.5) $$G_\Gamma, \Gamma' \vdash \mathsf{let}\ y : \mathcal{V}^M[\![\tau]\!] \Leftarrow \mathcal{C}^M[\![e]\!]\ \mathsf{in}\ (\mathsf{ref}_M\ \mathcal{V}^M[\![\tau]\!])\ y : \mathsf{M}\ (\mathsf{ref}\ \mathcal{V}^M[\![\tau]\!])$$

which yields the claim by $\mathsf{M}\ (\mathsf{ref}\ \mathcal{V}^M[\![\tau]\!]) \equiv \mathcal{C}^M[\![(\mathsf{ref}\ \tau)]\!]$.

- $\dfrac{\Gamma \vdash_{SML} e : \mathsf{ref}\ \tau}{\Gamma \vdash_{SML} !\, e : \tau}$. By induction,

(E.6) $$G_\Gamma, \Gamma' \vdash \mathcal{C}^M[\![e]\!] : \mathcal{C}^M[\![\mathsf{ref}\ \tau]\!]$$

By Lemma 1, rule (application), and rule (weakening),

(E.7) $$G_\Gamma, \Gamma', y : \mathcal{V}^M[\![\mathsf{ref}\ \tau]\!] \vdash !_M\ \mathcal{V}^M[\![\tau]\!] : \mathsf{ref}\ \mathcal{V}^M[\![\tau]\!] \to \mathsf{M}\ \mathcal{V}^M[\![\tau]\!]$$

By rule (application) applied to (E.7) and $y : \mathcal{V}^M[\![\mathsf{ref}\ \tau]\!]$,

(E.8) $$G_\Gamma, \Gamma', y : \mathcal{V}^M[\![\mathsf{ref}\ \tau]\!] \vdash (!_M\ \mathcal{V}^M[\![\tau]\!])\ y : \mathsf{M}\ \mathcal{V}^M[\![\tau]\!]$$

Finally, apply rule (let) to (E.6) and (E.8)

(E.9) $$G_\Gamma, \Gamma' \vdash \mathsf{let}\ y : \mathcal{V}^M[\![\mathsf{ref}\ \tau]\!] \Leftarrow \mathcal{C}^M[\![e]\!]\ \mathsf{in}\ (!_M\ \mathcal{V}^M[\![\tau]\!])\ y : \mathsf{M}\ \mathcal{V}^M[\![\tau]\!]$$

which yields the claim by $\mathsf{M}\ \mathcal{V}^M[\![\tau]\!] \equiv \mathcal{C}^M[\![\tau]\!]$.

- $\dfrac{\Gamma \vdash_{SML} e_1 : \mathsf{ref}\ \tau \quad \Gamma \vdash_{SML} e_2 : \tau}{\Gamma \vdash_{SML} e_1 := e_2 : ()}$. By induction,

(E.10) $$G_\Gamma, \Gamma' \vdash \mathcal{C}^M[\![e_1]\!] : \mathcal{C}^M[\![\mathsf{ref}\ \tau]\!]$$

and

(E.11) $$G_\Gamma, \Gamma' \vdash \mathcal{C}^M[\![e_2]\!] : \mathcal{C}^M[\![\tau]\!]$$

By Lemma 1, rule (application), and rule (weakening),

(E.12) $$G_\Gamma, \Gamma', y_1 : \mathcal{V}^M[\![\mathsf{ref}\ \tau]\!], y_2 : \mathcal{V}^M[\![\tau]\!] \vdash :=_M\ \mathcal{V}^M[\![\tau]\!] : \mathsf{ref}\ \mathcal{V}^M[\![\tau]\!] \to \mathcal{V}^M[\![\tau]\!] \to \mathsf{M}\ ()$$

Use rule (application) twice to get

(E.13) $$G_\Gamma, \Gamma', y_1 : \mathcal{V}^M[\![\mathsf{ref}\ \tau]\!], y_2 : \mathcal{V}^M[\![\tau]\!] \vdash (:=_M\ \mathcal{V}^M[\![\tau]\!])\ y_1\ y_2 : \mathsf{M}\ ()$$

Apply rule (let) to (E.11) and (E.13),

(E.14) $$G_\Gamma, \Gamma', y_1 : \mathcal{V}^M[\![\mathsf{ref}\ \tau]\!] \vdash \mathsf{let}\ y_2 : \mathcal{V}^M[\![\tau]\!] \Leftarrow \mathcal{C}^M[\![e_2]\!]\ \mathsf{in}\ (:=_M\ \mathcal{V}^M[\![\tau]\!])\ y_1\ y_2 : \mathsf{M}\ ()$$

Apply rule (let) to (E.10) and (E.14) to obtain the claim

$$G_\Gamma, \Gamma' \vdash \mathsf{let}\ y_1 : \mathcal{V}^M[\![\mathsf{ref}\ \tau]\!] \Leftarrow \mathcal{C}^M[\![e_1]\!]\ \mathsf{in}$$

$$\mathsf{let}\ y_2 : \mathcal{V}^M[\![\tau]\!] \Leftarrow \mathcal{C}^M[\![e_2]\!]\ \mathsf{in}\ (:=_M\ \mathcal{V}^M[\![\tau]\!])\ y_1\ y_2 : \mathsf{M}\ ()$$

where $\mathsf{M}\ () \equiv \mathcal{C}^M[\![()]\!]$.

■

F Operational Semantics of F_ω

The small-step operational semantics in this section are taken from work by Harper and Lillibridge [22,24]. Following Felleisen and Hieb [15], each of them defines a set of values v, a set of evaluation contexts E, and a set of reduction rules to be applied in an evaluation context.

F.1 Standard Call-by-Name

$$\begin{array}{ll} \text{values} & v ::= \lambda x:\sigma.e \mid \Lambda\alpha:\kappa.e \\ \text{evaluation contexts} & E ::= [\,] \mid E@e \mid E[\sigma] \end{array}$$

$$\begin{array}{l} E[(\lambda x:\sigma.e_1)@e_2] \rightarrow_{std-cbn} E[e_1\{x := e_2\}] \\ E[(\Lambda\alpha:\kappa.e)[\sigma]] \rightarrow_{std-cbn} E[e\{\alpha := \sigma\}] \end{array}$$

F.2 Standard Call-by-Value

$$\begin{array}{ll} \text{values} & v ::= x \mid \lambda x:\sigma.e \mid \Lambda\alpha:\kappa.e \\ \text{evaluation contexts} & E ::= [\,] \mid E@e \mid v@E \mid E[\sigma] \end{array}$$

$$\begin{array}{l} E[(\lambda x:\sigma.e)@v] \rightarrow_{std-cbv} E[e\{x := v\}] \\ E[(\Lambda\alpha:\kappa.e)[\sigma]] \rightarrow_{std-cbv} E[e\{\alpha := \sigma\}] \end{array}$$

F.3 ML-style Call-by-Value

$$\begin{array}{ll} \text{values} & v ::= x \mid \lambda x:\sigma.e \mid \Lambda\alpha:\kappa.v \\ \text{evaluation contexts} & E ::= [\,] \mid E@e \mid v@E \mid \Lambda\alpha:\kappa.E \mid E[\sigma] \end{array}$$

$$\begin{array}{l} E[(\lambda x:\sigma.e)@v] \rightarrow_{ml-cbv} E[e\{x := v\}] \\ E[(\Lambda\alpha:\kappa.v)[\sigma]] \rightarrow_{ml-cbv} E[v\{\alpha := \sigma\}] \end{array}$$

Electronic Notes in Theoretical Computer Science 10 (1998)
URL: http://www.elsevier.nl/locate/entcs/volume10.html 12 pages

Adapting Big-Step Semantics to Small-Step Style: Coinductive Interpretations and "Higher-Order" Derivations

Husain Ibraheem and David A. Schmidt [1]

Computing and Information Sciences Department
Kansas State University
Manhattan, KS 66506 USA.

Abstract

We adapt Kahn-style ("big-step") natural semantics to take on desirable aspects of small-step and denotational semantics forms, more precisely: *(i)* the ability to express divergent computations; *(ii)* the ability to reason about the (length of a) computation of a derivation; and *(iii)* the ability to compute upon and reason about higher-order values. To accomplish these results, we extend the classical, inductive interpretation of natural semantics with coinduction mechanisms and use "negative" rules to express divergence. A simple reformatting of the syntax of derivations allows a simple description of the "length" of a derivation. Finally, the recoding of closure values into denotational-semantics-like functions lets one embed derivations within closures that embed within derivations; in this sense, the semantics becomes "higher order." Examples are given to support the definitional developments.

Kahn-style ("big-step") natural semantics [9] captures a program's entire computation within a single derivation. This representation is more compact than what one obtains from state-transition, "small-step," and denotational definitions; in addition, natural semantics provides several additional attractions:

- When the natural semantics is defined in "environment form," that is, the semantics' propositions are ternary relations of the form, $\rho \vdash e \Downarrow v$, where $\rho \in \mathit{Environment}$, $e \in \mathit{Expression}$, and $v \in \mathit{Value}$, the resulting derivations are (almost always) *semicompositional.* That is, when one draws a physical derivation tree starting from an initial source program e_0 and initial (empty) environment ρ_0, then every node within the tree has the form $\rho' \vdash e' \Downarrow v'$, where e' is a subphrase of e_0. See Figure 1 for an example.

[1] Ph: +1-785-532-6350. schmidt@cis.ksu.edu. Ibraheem is supported by a Kuwait University scholarship; Schmidt is supported by NSF CCR-9633388.

Let $\rho_3 = \rho \oplus \{x \mapsto 3\}$ in the following:

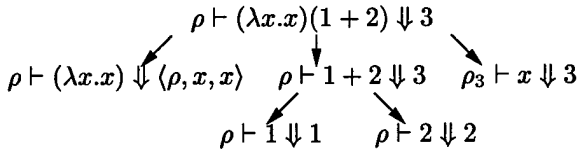


Fig. 1. A semicompositional derivation tree

- One can prove concisely properties of convergent derivations by induction on the height of the derivation tree.
- Static analyses are simple to formalize. The key concept, a program's *collecting semantics* [2,13,15], is defined as a simple function of the program's derivation: Given a program, e_0, its initial environment, ρ_0, and its resulting derivation tree, d, define the collecting semantics of subphrase e of e_0 in d to be $Collecting(d, e) = \{\rho \mid \rho \vdash e \Downarrow v \text{ is a node in } d\}$.

Alas, natural semantics is imperfect: One cannot reason about divergent programs, and its inherent first-ordered-ness makes natural semantics a poor choice for analysis of "open" or "modular" programs. Both denotational and small-step semantics hold advantages in these regards.

In this paper, we modify natural semantics to take on attractive aspects of small-step and denotational semantics without losing its attractive features. The features we wish to gain are

- One should be able to express divergent computations.
- One should be able to reason in terms of the length of a computation, whether the computation is convergent or divergent.
- One should be able to reason about "higher-order" values, e.g., closures.

We achieve our goals for a subset of natural semantics definitions that we call L-attributed—that is, left-to-right processing. L-attributed natural semantics definitions have a natural notion of sequential computation from left-to-right that resembles the computations done with a small-step semantics or with a Prolog interpreter applied to the natural semantics rule schemes [4]. In addition, we can discuss the "length" of a computation.

To gain divergent computations, we generate sets of "positive" (convergent) and "negative" (divergent) rules from the natural semantics rule schemes [3,14,13] and apply coinduction definition techniques to the negative rules. Finally, we modify the structure of higher-order values—closures—so that instead of being inert values, closures become "alive" by containing partial derivations. This allows denotational-semantics-like, extensional-style reasoning and opens the door to the application of partial evaluation techniques [5,8] to natural semantics definitions [7].

1 L-attributed Natural Semantics

Recall that an L-attributed attribute grammar [10] is a Knuthian attribute grammar where information flows from left to right; thus, the attributes for a parse tree can be calculated in a single, left-to-right, tree traversal.

Analogously, we define an *L-attributed*, environment-based, natural-semantics rule scheme as follows. A scheme of the form

$$\frac{\rho_1 \vdash e'_1 \Downarrow v_1 \quad \rho_2 \vdash e'_2 \Downarrow v_2 \quad \cdots \quad \rho_m \vdash e'_m \Downarrow v_m}{\rho_0 \vdash op(e_1, e_2, ..., e_n) \Downarrow v_{m+1}}$$

is L-attributed if

- the value of each ρ_i is a function of $e_1, e_2, ..., e_n$, and those ρ_j and v_j such that $j < i$
- the value of each e'_i is a function of $e_1, e_2, ..., e_n$, and those ρ_j such that $j \leq i$ and those v_j such that $j < i$
- the value of v_{m+1} is a function of $e_1, e_2, ..., e_n$, and those ρ_j and v_j such that $j < m + 1$

Again, the intuition is that a derivation with an L-attributed rule can be formulated by a left-to-right tree traversal, where the ρ_i-values are the "inherited attributes" and the v_i-values are the "synthesized attributes." But the rule scheme is a true attribute-grammar rule only if $m = n$ and $e'_i = e_i$, for all $1 \leq i \leq m$. This need not be the case—the standard rule for function application is a counterexample:

$$\frac{\rho \vdash e_1 \Downarrow \langle \rho', x', e' \rangle \quad \rho \vdash e_2 \Downarrow v_2 \quad \rho' \oplus \{x' \mapsto v_2\} \vdash e' \Downarrow v}{\rho \vdash e_1\ e_2 \Downarrow v}$$

Virtually every natural semantics definition in common use is L-attributed.

Assuming that no ρ_i or v_{m+1} manufactures new source program syntax, it is straightforward to prove that a set of L-attributed rule schemes forms a semicompositional definition. (Indeed, L-attribution is unimportant to the proof.)

2 Positive and Negative Rules

Although it is traditional to take an inductive interpretation of natural semantics rule schemes, we desire derivations for divergent programs, so we interpret each L-attributed rule scheme as inducing *positive* and *negative* rules, in the sense of Cousot and Cousot [3,13]: An L-attributed rule scheme induces a family of *positive* rules of the form

$$+ : \frac{\rho_1 \vdash e'_1 \Downarrow v_1 \quad \rho_2 \vdash e'_2 \Downarrow v_2 \quad \cdots \quad \rho_m \vdash e'_m \Downarrow v_m}{\rho_0 \vdash op(e_1, e_2, ..., e_n) \Downarrow v_{m+1}}$$

and m families of *negative* rules of the form

$$- : \frac{\rho_1 \vdash e'_1 \Downarrow v_1 \quad \rho_{k-1} \vdash e'_{k-1} \Downarrow v_{k-1} \quad \rho_k \vdash e_k \Uparrow}{\rho_0 \vdash op(e_1, ..., e_n) \Uparrow} \quad \text{for all } k \in 1..m$$

If desired, you can read the above as generating one positive rule scheme and m negative rule schemes, but it is traditional to generate the set of substitution instances (*rules*) of a rule scheme, hence we speak of one family of positive rules and m families of negative rules.

The positive rules derive convergent computations and the negative rules derive divergent ones. We define the *convergent derivations* to be the inductive (least-fixed point) interpretation of the positive rules.[2] Next, we define the *divergent derivations* by using the convergent derivations plus a coinductive (greatest fixed-point) interpretation of the negative rules.[3] The set of well-formed derivations, *wfd*, of the natural semantics is the union of the convergent and divergent derivations.

With the inductive interpretation of the positive rules comes inductive reasoning principles, most notably induction on the height of the convergent derivations, and with the coinductive interpretation comes coinductive reasoning.

3 Modifying the Rules into a "Smaller-Step" Format

A user of a natural semantics starts with an initial environment, program pair, ρ_0, e_0, and wants to draw a derivation. When she starts, the user has no clue whether a convergent derivation, $\rho_0 \vdash e_0 \Downarrow v$, or a divergent derivation, $\rho_0 \vdash e_0 \Uparrow$, will develop. The usual hack is to employ a Prolog-style "logical variable," V, to denote either of $\Downarrow v$ or $\Uparrow$, and start writing a derivation with a root of form $\rho_0 \vdash e_0 \Downarrow V$. But this is an imperfect start and fails to accurately deal with divergence—a formalization should do better.

We make the big-step, natural semantics into a "smaller step" one—based on the intuition that the triple, $\rho_0 \vdash e_0 \Downarrow v$, asserts the existence of a finite derivation of small steps, $(\rho_0 \vdash e_0) \rightarrow^* v$, we propose this new grammar for "raw derivations":

$$d \in \mathit{Derivation}$$

$$s \in \mathit{Sequent} \ \ (\rho \vdash e)$$

$$v \in \mathit{Value} \text{ (Note: assume } \mathit{Sequent} \cap \mathit{Value} = \emptyset)$$

$$d \ ::= \ v \ \mid \ \overset{s}{\underset{d_1 \quad \cdots \quad d_n}{\swarrow \quad \searrow}}, \ n \geq 0$$

That is, a derivation[4] is a value (e.g., an integer or a closure), or a tree whose root is a sequent, s (that is, $\rho \vdash e$), and whose subtrees are derivations.[5] If a sequent, s, has a sequence of computation steps that converge at v, then

[2] The L-attributed positive rules are continuous.

[3] The L-attributed negative rules are cocontinuous.

[4] Starting with a universe of finitely branching trees of at most countably infinite depth, we take the coinductive definition of the grammar rule for *Derivation*.

[5] We will also write this in the linear form, $s(d_1, ..., d_n)$.

Let $\rho_3 = \rho \oplus \{x \mapsto 3\}$ in the following:

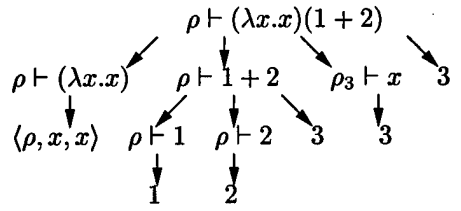


Fig. 2. Sample derivation in modified syntax

the corresponding derivation will have s at the root and have as its rightmost child subtree the leaf, v. Figure 2 displays a small example: Such trees bear a relationship to the ones derived with Plotkin-style, "small-step," structural-operational semantics rules [11], once one notes that the arrows from sequent to sequent represent "congruence" steps whereas the arrows from sequent to value represent "$\beta\delta$" steps. Indeed, based on this intuition, one can mechanically reformat the derivation tree into a Plotkin-style derivation by traversing and disassembling the derivation from left to right. This idea provides the intuition for the development in the next section.

Importantly, a divergent program will have a derivation with an infinite path; specifically, the derivation's rightmost path will be infinite, because the negative rules were formulated from L-attributed rule schemes.

From this point onwards, we work with the syntax of derivations defined above, and we use $wfd \subseteq \mathit{Derivation}$ to denote the set of well-formed derivations under the new syntax.

Here is some additional useful metanotation: We use $\rho \vdash e \downarrow v$ and $\rho \vdash e \uparrow$ as follows:

- $\rho \vdash e \downarrow v$ denotes a derivation, d, such that $root(d) = \rho \vdash e$ and the rightmost subtree of d is the leaf, v
- $\rho \vdash e \uparrow$ denotes a derivation, d, such that $root(d) = \rho \vdash e$ and the rightmost subtree of d is *not* a value.

This metanotation gives us a trivial representation of the new formats of positive and negative rules. Given this natural semantics rule scheme,

$$\frac{\rho_1 \vdash e_1' \Downarrow v_1 \quad \rho_2 \vdash e_2' \Downarrow v_2 \quad \cdots \quad \rho_m \vdash e_m' \Downarrow v_m}{\rho_0 \vdash op(e_1, e_2, ..., e_n) \Downarrow v_{m+1}}$$

the positive rules we induce have the form

$$+ : \frac{\rho_1 \vdash e_1' \downarrow v_1 \quad \rho_2 \vdash e_2' \downarrow v_2 \quad \cdots \quad \rho_m \vdash e_m' \downarrow v_m \quad v_{m+1}}{\rho_0 \vdash op(e_1, e_2, ..., e_n)}$$

and the negative rules now appear

$$- : \frac{\rho_1 \vdash e_1' \downarrow v_1 \quad \rho_{k-1} \vdash e_{k-1}' \downarrow v_{k-1} \quad \rho_k \vdash e_k \uparrow}{\rho_0 \vdash op(e_1, ..., e_n)}$$

for all $k \in 1..m$. Remember that $\rho \vdash e \downarrow v$ and $\rho \vdash e \uparrow$ describe deriva-

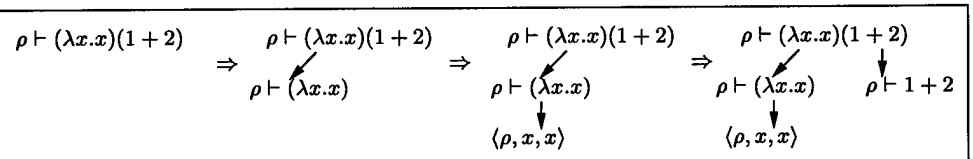


Fig. 3. The first four stages of a development

tions (not just sequents), therefore the rules resemble Prawitz-style natural deduction rules [12], where antecedents are derivation trees and succedents are sequents/propositions.

It is easy to prove that the set of derivations of the modified rules is isomorphic to the set of derivations of the original rules.

4 Derivation Discovery

The previous two sections defined the "semantics of natural semantics," namely, how the original natural semantics rule schemes induce families of positive and negative rules and how the rules are used to define *wfd*, the set of well-formed derivations.

But users use rule schemes to *compute* derivations—starting from an initial sequent, $\rho_0 \vdash e_0$, a user computes, step by step, a derivation tree. Using the new syntax of derivation trees, defined in the previous section, the process of derivation discovery becomes simple and natural: One uses the original, L-attributed natural semantics rule schemes to draw a derivation in incremental steps as a left-to-right tree traversal—there is no need for Prolog-style "logical variables." To state this more precisely, an original, L-attributed rule scheme,

$$\frac{\rho_1 \vdash e_1' \Downarrow v_1 \quad \rho_2 \vdash e_2' \Downarrow v_2 \quad \cdots \quad \rho_m \vdash e_m' \Downarrow v_m}{\rho_0 \vdash op(e_1, e_2, ..., e_n) \Downarrow v_{m+1}}$$

directs the traversal as follows:

(i) Starting from the "goal sequent," $\rho_0 \vdash op(e_1, e_2, ..., e_n)$, generate an arc to the "subgoal," $\rho_1 \vdash e_1'$; attempt to "satisfy" the subgoal by drawing a convergent derivation, $\rho_1 \vdash e_1' \downarrow v_1$.

(ii) Once the subgoal is achieved, repeatedly generate and attempt to satisfy subgoals $\rho_i \vdash e_i'$, for $1 < i \leq m$.

(iii) When all subgoals are achieved, compute v_{m+1} and attach this to the derivation as the rightmost child of the root, $\rho_0 \vdash op(e_1, e_2, ..., e_n)$.

For example, the discovery of the derivation in Figure 2 would start with the steps drawn in Figure 3 and would proceed in similar increments until the tree in Figure 2 results. We call a sequence of such partial derivation trees a *development.* Informally stated, a development is *complete* if while working from left to right, one reaches a derivation that belongs to *wfd.* A finite development might not be complete, and a complete development might not be finite: In the first case, an error in the source program, e.g., $\rho_0 \vdash 2 + true$,

can cause an unwanted termination in the development.

In the second case, this rule scheme, $\frac{\rho \vdash loop \Downarrow v}{\rho \vdash loop \Downarrow v}$, would of course generate this positive and this negative rule:

$$+ : \frac{\rho \vdash loop \downarrow v \quad v}{\rho \vdash loop} \qquad - : \frac{\rho \vdash loop \uparrow}{\rho \vdash loop}$$

Using these rules, we obtain from the starting sequent, $\rho_0 \vdash loop + 1$, this infinite derivation

$$\rho_0 \vdash loop + 1 \longrightarrow \rho_0 \vdash loop \longrightarrow \rho_0 \vdash loop \longrightarrow \cdots$$

The derivation is well formed because the rule scheme for arithmetic addition, namely, $\frac{\rho \vdash e_1 \Downarrow v_1 \quad \rho \vdash e_2 \Downarrow v_2}{\rho \vdash e_1 + e_2 \Downarrow plus(v_1, v_2)}$, generates positive rules of this form

$$+ : \frac{\rho \vdash e_1 \downarrow v_1 \quad \rho \vdash e_2 \downarrow v_2 \quad plus(v_1, v_2)}{\rho \vdash e_1 + e_2}$$

and negative rules of these two forms

$$- : \frac{\rho \vdash e_1 \uparrow}{\rho \vdash e_1 + e_2} \qquad - : \frac{\rho \vdash e_1 \downarrow v_1 \quad \rho \vdash e_2 \uparrow}{\rho \vdash e_1 + e_2}$$

and the first form of negative rule justifies that the derivation is well formed, that is, it is a member of *wfd*.

Figure 4 gives formalizations of when a raw derivation is partially developed and is completely developed. An important consequence of the definitions is that left-to-right derivation discovery generates a sequence of partially developed trees. When the sequence completes, the result is a complete development.

4.1 Semantics of Developments

Figure 4 defines the crucial aspects of developments, and several easily proved consequences follow. First, here is a simple semantics of developments: Given a development, $d_0 \Rightarrow d_1 \Rightarrow \cdots \Rightarrow d_i \Rightarrow \cdots$, define the set of derivations denoted by a partial derivation, d_i, to be $S_i = \{d \in wfd \mid d \text{ extends } d_i\}$. That is, each d_i defines the set of those well-formed derivations whose prefix is d_i. For all $j > i$, $S_j \subseteq S_i$. The semantics of a complete development is $S_\omega = \bigcap_{i \geq 0} S_i$.

With this semantics, we have easy proofs of

- *soundness:* every completely developed derivation is well formed, that is, it is a member of *wfd*
- *adequacy:* every $d \in wfd$ has a complete development

Soundness follows because the limit of a complete development is a derivation that belongs to S_ω; adequacy holds because the natural semantics is L-attributed, hence every well-formed derivation must have a complete development that is a sequence of partial derivations drawn in a left-to-right traversal.

$isPartial(v)$

$isPartial(s(d_1, ..., d_k))$ iff $0 \leq k \leq m$ and there exists

$\pm : \dfrac{d_1 \cdots d_k \quad d_{k+1} \cdots d_m}{s}$

such that for all $0 < j < k$, $isComplete^+(d_j)$, and $isPartial(d_k)$

(Take greatest solution of this definition.)

$isComplete^+(v)$

$isComplete^+(s(d_1, ..., d_m))$ iff there exists $+ : \dfrac{d_1 \cdots d_m}{s}$

such that for all $0 < j \leq m$, $isComplete^+(d_j)$

(Take least solution of this definition.)

$isComplete(v)$

$isComplete(s(d_1, ..., d_m))$ iff there exists $\pm : \dfrac{d_1 \cdots d_m}{s}$

such that for all $0 < j < m$, $isComplete^+(d_j)$, and $isComplete(d_m)$

(Take greatest solution of this definition.)

Fig. 4. Definitions of partially and completely developed derivations

Hence the set of well-formed derivations, *wfd*, is characterized by the set of complete developments. As a result, one obtains a small-step-semantics-like induction principle: induction on the length of a complete development.

5 Higher-Order Natural Semantics

Traditional natural semantics is "too first ordered"—values like closures, $\langle \rho, x, e \rangle$, are inert, "dumb," data structures. In this section, we make closures alive and "smart" by modifying them so that they contain raw derivations—a closure, $\langle \rho, x, e \rangle$ is reformatted into $\boldsymbol{\lambda} A.\ \rho \oplus \{x \mapsto A\} \vdash e$, that is, a mapping from a value, A, to a sequent that uses A. Of course, the sequent is a raw derivation, so we generalize the idea and allow a closure to hold a partially developed derivation, d, that is, $\boldsymbol{\lambda} A.\ d$, such that $root(d) = \rho \oplus \{x \mapsto A\} \vdash e$.

Using this formulation, we can develop the bodies of closures and we can apply techniques akin to those from denotational semantics to reason about the behavior of a closure prior to its application.

$$\rho \vdash x \Downarrow \rho_x \qquad \rho \vdash \lambda x.\, e \Downarrow \boldsymbol{\lambda} A.\, (\rho \oplus \{x \mapsto A\} \vdash e)^*$$

where s^* denotes a derivation, d, such that $root(d) = s$ and $isPartial(d)$

$$\frac{\rho \vdash e_1 \Downarrow \boldsymbol{\lambda} A.\, d \quad \rho \vdash e_2 \Downarrow v_2 \quad [v_2/A]d \Downarrow v}{\rho \vdash e_1\ e_2 \Downarrow v}$$

Fig. 5. Natural semantics rule schemes for higher-order closures

Let $\rho_\alpha = \rho \oplus \{x \mapsto \alpha\}$ in the following:

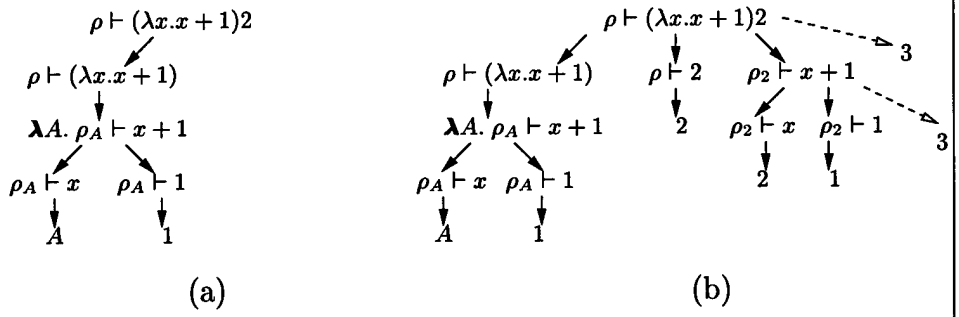


Fig. 6. An example with speculative development

The syntax we use goes as follows:

$d \in \mathit{Derivation}$

$s \in \mathit{Sequent}\ \ (\rho \vdash e)$

$v \in \mathit{Value}$ (Note: assume $\mathit{Sequent} \cap \mathit{Value} = \emptyset$)

$A \in \mathit{VariableValue}$

$p \in \mathit{PrimitiveFirstOrderValue}$ (e.g., integer)

$$d \ ::=\ v \ \mid\ \overset{s}{\underset{d_1 \cdots d_n}{\diagdown\ \diagup}},\ n \geq 0$$

$$v \ ::=\ p \ \mid\ A \ \mid\ \boldsymbol{\lambda} A.\, d$$

This syntax is "higher order" in the sense that derivations can contain values that can contain derivations.

Figure 5 presents a set of natural semantics rule schemes for the new form of closures. The rule schemes in Figure 5 permit a development to proceed within a closure value; we call this a *speculative development.* For example, Figure 6(a) shows a partial development of a program where the value part for $\lambda x.\ x + 1$ is a closure that contains speculative development. This speeds up the the remainder of the development, because the speculative development can be used with the substitution of 2 for A, as seen in Figure 6(b); this quickly leads to completion of the derivation, as indicated by the two dotted arrows in Figure 6(b).

Of course, a serious application of speculative development would not limit

itself to a strict left-to-right strategy; examples like $\rho \vdash \lambda x.\ (x + 1) + (2 + 3)$ and $\rho \vdash \lambda x.\ \texttt{if}\ x\ (\lambda y.\ x)\ (\lambda x.\ x)$ can benefit from a non-leftmost speculative development.

Indeed, partial evaluation [5,8] applied to natural semantics derivations is aggressive speculative development, and as an ongoing project, we have been applying partial evaluation to the variant of natural semantics seen in this paper to uncover the feasibility of automated, static analysis of open ("modular") programs [7].

6 Applications, Related Work, and Conclusions

As noted above, the modifications of natural semantics derivations let one work easily with partial derivations and derivations of open programs (source programs containing unbound variables). Partial evaluation techniques now apply directly to natural semantics framework, as demonstrated by the ongoing Ph.D. research of Ibraheem [7], where a supercompilation algorithm by Glück and Sørensen has been adapted to operate on partial derivations in place of so-called "process trees" [5]. This line of research aims to show that static analysis of program modules can be performed with the same partial evaluation machinery used for complete programs.

Of course, the intuition that a natural semantics derivation can be computed by a Prolog interpreter is a standard one. Deransart and Ferrand [4] give a careful formulation of how a goal tree can be computed by a Prolog interpretater in a left-to-right, incremental fashion. Gunter and Remy [6] adapt Prolog goal search to natural semantics definitions; they define a typed notation, RAVL, for their presentation and show how one might prove a type-safety property in their framework. And Berry's Ph.D. thesis describes a suite of tools for drawing "animations" of the incremental generation of partial derivations [1]. Most recently, Stoughton has studied the formalizaton of partial generation of derivations within resumption semantics [16].

Based on the investigations in this paper, one must conclude that the relationships between big-step, small-step, and denotational semantics are closer, in a formal sense, than what has been demonstrated so far in the literature. Although it is premature to speculate exactly what class of semantic definition is simultaneously both small-step- and big-step-like, it is clear that there is a subclass of operational semantics definitions that form an intersection between the two formats. And, the connections between denotational and operational semantics need closer examination as well.

References

[1] D. Berry. *Generating Program Animators from Programming Language Semantics.* PhD thesis, Edinburgh University, 1991.

[2] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs. In *Proc. 4th ACM Symp. on Principles of Programming Languages*, pages 238–252. ACM Press, 1977.

[3] P. Cousot and R. Cousot. Inductive definitions, semantics, and abstract interpretation. In *Proc. 19th ACM Symp. on Principles of Programming Languages*, pages 83–94. ACM Press, 1992.

[4] P. Deransart and G. Ferrand. An operational formal definition of PROLOG. In *Proc. IEEE Symp. on Logic Programming*, pages 162–172. IEEE Press, 1987.

[5] R. Glück and M. H. Sørensen. A roadmap to metacomputation by supercompilation. In O. Danvy, R. Glück, and P. Thiemann, editors, *Partial Evaluation*, number 1110 in Lecture Notes in Computer Science, pages 137–160. Springer-Verlag, 1996.

[6] C. Gunter and D. Rémy. A proof-theoretic assessment of runtime type errors. Technical memo 11261-921230-43TM, AT&T Bell Laboratories, Murray Hill, PA, 1993.

[7] H. Ibraheem and D. Schmidt. Partial evaluation of higher-order natural-semantics derivations. In M. Leuschel, editor, *Proc. Workshop on Specialization of Declarative Programs and its Applications, Port Jefferson, NY.* `http://www.cis.ksu.edu/~schmidt/papers/driving.ps.Z`, 1997.

[8] N.D. Jones, C. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation.* Prentice-Hall, 1993.

[9] G. Kahn. Natural semantics. In *Proc. STACS '87*, pages 22–39. Lecture Notes in Computer Science 247, Springer, Berlin, 1987.

[10] P. Lewis, D. Rosenkrantz, and R. Stearns. *Compiler Design Theory.* Addison-Wesley, 1978.

[11] Gordon D. Plotkin. A structural approach to operational semantics. Technical Report FN-19, Computer Science Department, Aarhus University, Aarhus, Denmark, September 1981.

[12] D. Prawitz. *Natural Deduction: A Proof-Theoretical Study.* Almqvist and Wiksell, Stockholm, 1965.

[13] D.A. Schmidt. Trace-based abstract interpretation of operational semantics. *J. Lisp and Symbolic Computation.* In press. Available from `www.cis.ksu.edu/~schmidt/papers/aiosh.ps.Z`

[14] D.A. Schmidt. Natural-semantics-based abstract interpretation. In A. Mycroft, editor, *Static Analysis Symposium*, number 983 in Lecture Notes in Computer Science, pages 1–18. Springer-Verlag, 1995.

[15] D.A. Schmidt. Data-flow analysis is model checking of abstract interpretations. In *Proc. 25th ACM Symp. on Principles of Prog. Languages.* ACM Press, 1998.

[16] A. Stoughton. An Operational Semantics Framework Supporting the Incremental Construction of Derivation Trees. This proceedings.

Electronic Notes in Theoretical Computer Science 10 (1998)
URL: http://www.elsevier.nl/locate/entcs/volume10.html 12 pages

An Operational Semantics Framework Supporting the Incremental Construction of Derivation Trees

Allen Stoughton [1]

Department of Computing and Information Sciences
Kansas State University
Manhattan, KS 66506, USA
http://www.cis.ksu.edu/~allen/home.html

Abstract

We describe the current state of the design and implementation of Dops, a framework for Deterministic OPerational Semantics that will support the incremental construction of derivation trees, starting from term/input pairs. This process of derivation tree expansion may terminate with either a complete derivation tree, explaining why a term/input pair evaluates to a particular output, or with a blocked incomplete derivation tree, explaining why a term/input pair fails to evaluate to an output; or the process may go on forever, yielding, in the limit, an infinite incomplete derivation tree, explaining why a term/input pair fails to evaluate to an output.

The Dops metalanguage is a typed lambda calculus in which all expressions converge. Semantic rules are specified by lambda terms involving resumptions, which are used by a rule to consume the outputs of sub-evaluations and then resume the rule's work. A rule's type describes the number and kinds of sub-evaluations that the rule can initiate, and indicates whether the rule can block. The semantics of Dops is defined in an object language-independent manner as a small-step semantics on concrete derivation trees: trees involving resumptions. These concrete derivation trees can then be abstracted into ordinary derivation trees by forgetting the resumptions.

1 The incremental construction of derivation trees

We begin by defining the operational semantics that we will use as an example throughout the rest of the paper: a big-step, environment-based semantics of the untyped, call-by-value lambda calculus.

[1] Partially supported by NSF/DARPA under grant CCR-9633388.

$$\frac{n \in \mathsf{Int}}{\mathsf{Var}\, n \in \mathsf{Exp}} \qquad \frac{a, b \in \mathsf{Exp}}{\mathsf{App}\langle a, b\rangle \in \mathsf{Exp}} \qquad \frac{n \in \mathsf{Int} \quad a \in \mathsf{Exp}}{\mathsf{Lam}\langle n, a\rangle \in \mathsf{Exp}}$$

Fig. 1. Syntax of lambda expressions

$$\frac{e \in \mathsf{Env} \quad n \in \mathsf{Int} \quad a \in \mathsf{Exp}}{\mathsf{Clos}\langle e, n, a\rangle \in \mathsf{Vlu}}$$

$$\mathsf{Nil}\langle\,\rangle \in \mathsf{Env} \qquad \frac{n \in \mathsf{Int} \quad x \in \mathsf{Vlu} \quad e \in \mathsf{Env}}{\mathsf{Cons}\langle n, x, e\rangle \in \mathsf{Env}}$$

Fig. 2. Values and environments

The set Exp of *lambda expressions* is inductively defined by the rules of Figure 1, where Int is the set of all integers. We abbreviate

$$\mathsf{App}\langle\mathsf{Lam}\langle 0, \mathsf{App}\langle\mathsf{Var}\, 0, \mathsf{Var}\, 0\rangle\rangle, \mathsf{Lam}\langle 0, \mathsf{App}\langle\mathsf{Var}\, 0, \mathsf{Var}\, 0\rangle\rangle\rangle$$

to $(\lambda_0.\, v_0\, v_0)(\lambda_0.\, v_0\, v_0)$, and make use of similar abbreviations (in which v_n abbreviates $\mathsf{Var}\, n$, the nth variable) without further comment.

To define the semantics of expression evaluation, we need two simple semantic spaces. The sets Vlu of *values* and Env of *environments* are defined inductively by the rules of Figure 2. For example, if x and y are values, then

$$e = \mathsf{Cons}\langle 1, x, \mathsf{Cons}\langle 3, y, \mathsf{Nil}\langle\,\rangle\rangle\rangle$$

is an environment: the one in which variable 1 has value x, and variable 3 has value y. Note that e is sorted by variable number. The functions on environments that we define will assume and preserve the sortedness of environments. We also need the familiar auxiliary functions for looking up the value of an identifier in an environment and updating an environment to reflect a new binding:

$$\mathsf{Lookup}\colon \mathsf{Env} \to \mathsf{Int} \to (\{\langle\,\rangle\} + \mathsf{Vlu})$$
$$\mathsf{Update}\colon \mathsf{Env} \to \mathsf{Int} \to \mathsf{Vlu} \to \mathsf{Env}.$$

To understand the significance of the sum in Lookup's type, consider the environment e defined above. Then, $\mathsf{Lookup}\, e\, 2$ is $\mathsf{in}(0, \langle\,\rangle)$, the injection of the empty tuple into the 0th component of the sum, since variable 2 is not bound in e. And $\mathsf{Lookup}\, e\, 1$ is $\mathsf{in}(1, x)$, the injection of variable 1's value in e into the 1st component of the sum.

The semantics of expression evaluation can be defined as in Figure 3, where we read "$a, e \Rightarrow x$" as "expression a in environment e evaluates to value x". We consider the single premise of the variable evaluation rule to be a "side-condition", since it doesn't involve expression evaluation. The most straightforward way to view this definition is as the inductive definition of the relation $\Rightarrow \subseteq \mathsf{Exp} \times \mathsf{Env} \times \mathsf{Vlu}$, where $a, e \Rightarrow x$ abbreviates $\langle a, e, x\rangle \in \Rightarrow$. Given this interpretation, we can prove the following facts:

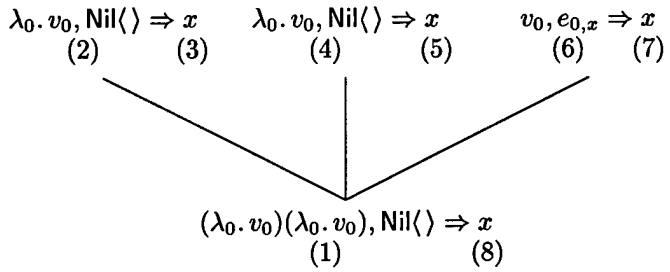
(i) $\Rightarrow$ is a partial function from $\mathsf{Exp} \times \mathsf{Env}$ to Vlu.

$$\frac{\mathsf{Lookup}\, e\, n = \mathsf{in}(1, x)}{\mathsf{Var}\, n, e \Rightarrow x}$$

$$\mathsf{Lam}\langle n, a\rangle, e \Rightarrow \mathsf{Clos}\langle e, n, a\rangle$$

$$\frac{a, e \Rightarrow \mathsf{Clos}\langle e', n, a'\rangle \quad b, e \Rightarrow y \quad a', \mathsf{Update}\, e'\, n\, y \Rightarrow z}{\mathsf{App}\langle a, b\rangle, e \Rightarrow z}$$

Fig. 3. Definition of expression evaluation



(where $x = \mathsf{Clos}\langle \mathsf{Nil}\langle\,\rangle, 0, v_0\rangle$ and $e_{0,x} = \mathsf{Cons}\langle 0, x, \mathsf{Nil}\langle\,\rangle\rangle$)

Fig. 4. Complete derivation

(ii) $(\lambda_0.\, v_0)(\lambda_0.\, v_0), \mathsf{Nil}\langle\,\rangle \Rightarrow \mathsf{Clos}\langle \mathsf{Nil}\langle\,\rangle, 0, v_0\rangle$.

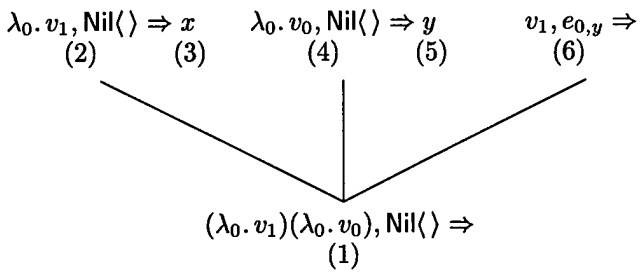
(iii) $(\lambda_0.\, v_1)(\lambda_0.\, v_0), \mathsf{Nil}\langle\,\rangle \Rightarrow x$ for no $x \in \mathsf{Vlu}$.

(iv) $(\lambda_0.\, v_0\, v_0)(\lambda_0.\, v_0\, v_0), \mathsf{Nil}\langle\,\rangle \Rightarrow x$ for no $x \in \mathsf{Vlu}$.

It is also possible to think about expression evaluation more concretely. For example, Figure 4 (ignore the labels (1)-(8) for now) consists of a derivation tree proving (providing evidence for) Fact (ii). Since the leftmost and middle children of this tree are instances of the axiom for abstraction evaluation, and the rightmost child follows by the variable rule (we omit the rule's premise, since it's a side-condition), the conclusion follows by the application rule.

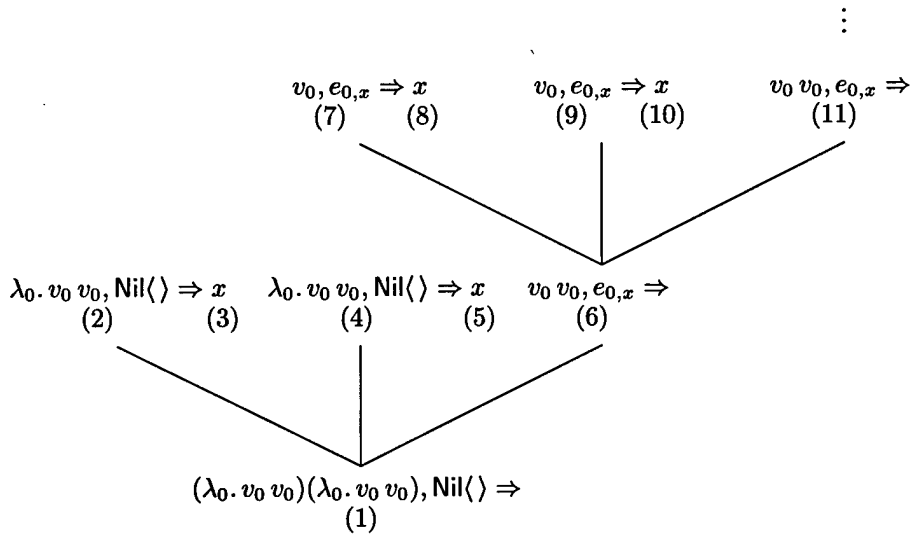
But, it is also possible and useful to think even more concretely, to focus on the step-by-step procedure in which derivation trees are constructed. With the tree of Figure 4, we begin, in Step (1), with the incomplete derivation tree consisting of the expression/environment pair $(\lambda_0.\, v_0)(\lambda_0.\, v_0), \mathsf{Nil}\langle\,\rangle$. Next, since our expression is an application, we begin evaluating the left side of the application, in Step (2), and finish this evaluation, in Step (3). In Steps (4) and (5), we evaluate the right side of the application. In Steps (6) and (7), we evaluate the expression of the closure x in the environment that is formed by binding the variable of the closure in the environment of the closure to the value of the right side of the application. Finally, in Step (8), we take the result of this rightmost evaluation and make it the result of our overall evaluation, giving us a complete derivation providing evidence for Fact (ii).

It is easy to prove that the tree expansion procedure that we followed above is sound and complete. If we start with an incomplete tree consisting



(where $x = \mathsf{Clos}\langle\mathsf{Nil}\langle\rangle, 0, v_1\rangle$, $y = \mathsf{Clos}\langle\mathsf{Nil}\langle\rangle, 0, v_0\rangle$ and $e_{0,y} = \mathsf{Cons}\langle 0, y, \mathsf{Nil}\langle\rangle\rangle$)

Fig. 5. Blocked incomplete derivation



(where $x = \mathsf{Clos}\langle\mathsf{Nil}\langle\rangle, 0, v_0\, v_0\rangle$ and $e_{0,x} = \mathsf{Cons}\langle 0, x, \mathsf{Nil}\langle\rangle\rangle$)

Fig. 6. Infinite incomplete derivation

of an expression/environment pair a, e and terminate with a complete tree whose root is $a, e \Rightarrow x$, then a, e evaluates to x. And, if a, e evaluates to x, then the procedure will turn the incomplete tree consisting of a, e into a complete tree with root $a, e \Rightarrow x$. When the procedure doesn't terminate with a complete derivation tree, it provides an explanation for why the starting expression/environment pair fails to evaluate to any value. Figure 5 gives an explanation for why Fact (iii) holds: the procedure terminates with a blocked incomplete derivation tree, since variable 1 is not bound in environment $e_{0,y}$. And Figure 6 gives an explanation for why Fact (iv) holds: the procedure fails to terminate, giving, in the limit, an infinite incomplete derivation tree.

```
sort Exp = Var of Int | App of {Exp, Exp} | Lam of {Int, Exp}

datatype Vlu = Clos of {Env, Int, Exp}
and      Env = Nil  of {} | Cons of {Int, Vlu, Env}
```

Fig. 7. Dops definitions of sorts and datatypes

2 A framework for deterministic operational semantics

We are designing and implementing a framework for Deterministic OPerational Semantics called Dops that will support the incremental construction of derivation trees, starting from object language term/input pairs. We restrict our attention to deterministic semantics for two reasons. First, one often wants a semantics to be deterministic, and so it is useful to have frameworks in which expressed semantics are guaranteed to be deterministic. Second, to be useful in practice, our tree expansion procedure itself will have to be deterministic, which means that any nondeterminism would have to be reflected either in the structure of the derivation trees themselves or in a lack of monotonicity of the tree expansion process. Neither of these alternatives seems desirable.

The operational semantics of an object language is expressed in the Dops metalanguage, a typed lambda calculus with sums, products, algebraic datatypes (recursive types not directly involving function types) and primitive recursion. This lambda calculus only expresses total functions, i.e., all well-typed lambda terms converge to values. In the metalanguage, n-ary sums, products and tuples are written like $[\sigma_0, \ldots, \sigma_{n-1}]$, $\{\sigma_0, \ldots, \sigma_{n-1}\}$ and $\{x_0, \ldots, x_{n-1}\}$, respectively, so that $\{\}$ is the single value of the unit type $\{\}$. Much of the metalanguage's syntax is reminiscent of Standard ML.

The syntactic categories of an object language are defined as sorts in the Dops metalanguage, and each sort has associated with it *input* and *output types*. Figure 7 shows how the single sort Exp of our example object language, along with its associated input and output types, Env and Vlu, can be expressed in our metalanguage. Figure 8 shows how the auxiliary functions Lookup and Update can be defined in the Dops metalanguage, using primitive recursion. (Straightforward constraints are used to prohibit general recursion.)

For each constructor of a given sort (Var, Lam and App in our example object language), a corresponding semantic rule must be specified as a metalanguage term. When an object language semantics would ordinarily have multiple inference rules for a single constructor, e.g., for a conditional operator, the multiple rules will have to be combined into a single metalanguage term, in a process that is similar to the "left-factoring" of [7]. The rule corresponding to a constructor δ of sort σ takes in an instance p of the constructor's data and a value x of σ's input type, and describes how the term $\delta\, p$ should be evaluated with input x. This evaluation may cause various sub-evaluations to be initiated, and may eventually terminate with the production of a value of σ's output type. Resumptions are used to consume the output values of

```
type VluOpt = [{}, Vlu]

rec Lookup : Env -> Int -> VluOpt =
      Nil{}         => fn _ : Int => in(0, VluOpt, {})
    | Cons{l, u, e} => fn n : Int =>
        case n < l of
             True{}  => in(0, VluOpt, {})
           | False{} =>
               case n = l of
                    True{}  => u
                  | False{} => Lookup e n
               esac
        esac

rec Update : Env -> Int -> Vlu -> Env =
      Nil{}         => fn n : Int => fn v : Int => Cons{n, v, Nil{}}
    | Cons{l, u, e} => fn n : Int => fn v : Int =>
        case n < l of
             True{}  => Cons{n, v, Cons{l, u, e}}
           | False{} =>
               case n = l of
                    True{}  => Cons{l, v, e}
                  | False{} => Cons{l, u, Update e n v}
               esac
        esac
```

Fig. 8. Dops definitions of auxiliary functions

sub-evaluations and then resume the rule's work. The types of rules involve *action types*, which describe the number and kinds of sub-evaluations that an application of a rule is capable of initiating, and indicate whether an application of a rule is capable of blocking. Since the metalanguage is deterministic, and there is only one rule per constructor, only deterministic semantics can be expressed in Dops.

Figure 9 shows how the semantic rules of our example object language can be expressed in the Dops metalanguage. Consider the most complex of these rules: App. The lambda term for App takes in the left and right sides, a and b, of the application to be evaluated, along with the environment e in which the evaluation should be carried out. It then returns an element of the action type AppAct. Action types always consist of sums with two or more components. Since the 0th component of AppAct is the empty type, we know that the evaluation of an application is incapable of immediately blocking; if it had been the unit type, then immediate blocking might have been possible. And, since the 1st component of AppAct is also the empty type, we know that the evaluation of an application cannot immediately result in a value of type Vlu (the output type of our constructor's sort); if this component had

```
type VarAct = [{}, Vlu]

rule Var : Int -> Env -> VarAct =
       fn n : Int => fn e : Env => Lookup e n

type AppAct3 = [[], Vlu]
type AppAct2 = [[], [], {Exp, Env, Vlu -> AppAct3}]
type AppAct1 = [[], [], {Exp, Env, Vlu -> AppAct2}]
type AppAct  = [[], [], {Exp, Env, Vlu -> AppAct1}]

rule App : {Exp, Exp} -> Env -> AppAct =
       fn {a, b} : {Exp, Exp} => fn e : Env =>
            in(2, AppAct,
               {a, e,
                fn x : Vlu =>
                     case x of
                          Clos{e', n, a'} =>
                            in(2, AppAct1,
                               {b, e,
                                fn y : Vlu =>
                                     in(2, AppAct2,
                                        {a', Update e' n y,
                                         fn z : Vlu =>
                                              in(1, AppAct3, z)})})
                     esac})

type LamAct = [[], Vlu]

rule Lam : {Int, Exp} -> Env -> LamAct =
       fn {n, a} : {Int, Exp} => fn e : Env =>
            in(1, LamAct, Clos{e, n, a})
```

Fig. 9. Dops definitions of semantic rules

been Vlu, then immediate production of an output value might have been possible. Thus the value returned by the application rule will have to consist of (the injection into the 2nd component of the sum of) a triple with type $\{\mathsf{Exp}, \mathsf{Env}, \mathsf{Vlu} \to \mathsf{AppAct1}\}$. The triple returned should be thought of as a request to initiate a sub-evaluation: to evaluate the 0th component of the triple with its 1st component as input, and then to supply the output value produced by this sub-evaluation to the resumption that is the 2nd component of the triple. The actual triple returned is thus a request to evaluate the left side a of the application in the environment e, and then to call the supplied resumption with the output value x of this sub-evaluation. The value x must be a closure, and the resumption first gives names to the components of the closure, and then initiates a second sub-evaluation: the evaluation of the right side b of the application in the environment e, where the output value y of the

$$\mathsf{Init}\langle a, x\rangle \in \Gamma \qquad \frac{\bar{\gamma} \in \Gamma^{+}}{\mathsf{Inc}\langle a, x, \bar{\gamma}, f\rangle \in \Gamma} \qquad \frac{\bar{\gamma} \in \Gamma^{*}}{\mathsf{Comp}\langle a, x, \bar{\gamma}, y\rangle \in \Gamma}$$

Fig. 10. Concrete derivation trees

sub-evaluation is to be given to the supplied resumption. This resumption, when invoked, will initiate a third and final sub-evaluation: the evaluation of the expression a' of the closure in the environment that is obtained by updating the environment e' of the closure so that the variable n of the closure is bound to the value y of b, where the output value z of this sub-evaluation is to be given to the final resumption, which must produce a value of action type AppAct3. Since AppAct3 has only two components, and only its 1st component is nonempty, this resumption must yield (the injection into the 1st component of AppAct3 of) an output value. The actual value returned is, of course, z.

By examining the action types VarAct and LamAct, we can tell that evaluating variables and lambda expressions never involves the initiation of sub-evaluations. In particular, the side-condition of the variable evaluation rule is handled inside the rule. According to VarAct, variable evaluation may be capable of blocking, since its 0th component is the unit type; and, if we look at the semantic rule for variable evaluation, we will see that variable evaluation blocks when a variable is looked up in an environment where it is unbound. On the other hand, LamAct tells us that lambda expression evaluation always terminates normally.

The semantics of Dops is defined in an object language-independent manner via a small-step semantics on the set Γ of *concrete derivation trees*, which are inductively defined in Figure 10. In this figure, Γ^* denotes the set of all tuples of elements of Γ, and Γ^+ denotes the set of all nonempty tuples of elements of Γ. When evaluating a term a with input x, one starts with the initial concrete derivation tree $\mathsf{Init}\langle a, x\rangle$. After some number of tree expansion steps, one may have an incomplete concrete derivation tree of the form $\mathsf{Inc}\langle a, x, \bar{\gamma}, f\rangle$. Here the elements of $\bar{\gamma}$ are the sub-derivations that have been constructed so far during the evaluation, and the resumption f is waiting for the last sub-derivation of $\bar{\gamma}$ to become complete; then the output value of this sub-derivation will be supplied to the resumption. Eventually, the tree expansion process may terminate with a complete concrete derivation tree of the form $\mathsf{Comp}\langle a, x, \bar{\gamma}, y\rangle$. Here, y is the output value obtained after evaluating a with input x, and $\bar{\gamma}$ would only be the empty tuple if the complete derivation tree was formed directly from $\mathsf{Init}\langle a, x\rangle$. There is a typing system for concrete derivation trees that puts some additional constraints on these trees, requiring the types of their components to be compatible and requiring all non-final sub-derivations to be complete.

The tree expansion relation $\rightarrow \subseteq \Gamma \times \Gamma$ is inductively defined by Figure 11, where $i \geq 0$, $\Downarrow$ is the metalanguage evaluation relation, rule_δ denotes the rule (a metalanguage term) corresponding to the constructor δ, and @ appends tuples. Note that the premises of the first four rules don't involve tree expansion and

$$\frac{\mathsf{rule}_\delta\, p\, x \Downarrow \mathsf{in}(1, \sigma, y)}{\mathsf{Init}\langle \delta\, p, x\rangle \to \mathsf{Comp}\langle \delta\, p, x, \langle\,\rangle, y\rangle}$$

$$\frac{\mathsf{rule}_\delta\, p\, x \Downarrow \mathsf{in}(i+2, \sigma, \{a, y, f\})}{\mathsf{Init}\langle \delta\, p, x\rangle \to \mathsf{Inc}\langle \delta\, p, x, \langle \mathsf{Init}\langle a, y\rangle\rangle, f\rangle}$$

$$\frac{f\, z \Downarrow \mathsf{in}(1, \sigma, w)}{\begin{array}{l}\mathsf{Inc}\langle a, x, \bar{\gamma}_1 \mathbin{@} \langle \mathsf{Comp}\langle b, y, \bar{\gamma}_2, z\rangle\rangle, f\rangle \to \\ \mathsf{Comp}\langle a, x, \bar{\gamma}_1 \mathbin{@} \langle \mathsf{Comp}\langle b, y, \bar{\gamma}_2, z\rangle\rangle, w\rangle\end{array}}$$

$$\frac{f\, z \Downarrow \mathsf{in}(i+2, \sigma, \{c, w, g\})}{\begin{array}{l}\mathsf{Inc}\langle a, x, \bar{\gamma}_1 \mathbin{@} \langle \mathsf{Comp}\langle b, y, \bar{\gamma}_2, z\rangle\rangle, f\rangle \to \\ \mathsf{Comp}\langle a, x, \bar{\gamma}_1 \mathbin{@} \langle \mathsf{Comp}\langle b, y, \bar{\gamma}_2, z\rangle, \mathsf{Init}\langle c, w\rangle\rangle, g\rangle\end{array}}$$

$$\frac{\gamma_1 \to \gamma_2}{\mathsf{Inc}\langle \delta\, p, x, \bar{\gamma} \mathbin{@} \langle \gamma_1\rangle, f\rangle \to \mathsf{Inc}\langle \delta\, p, x, \bar{\gamma} \mathbin{@} \langle \gamma_2\rangle, f\rangle}$$

Fig. 11. Definition of tree expansion relation

so can be viewed as side-conditions.

The first two tree expansion rules show how an initial concrete derivation tree $\mathsf{Init}\langle \delta\, p, x\rangle$ is expanded. We proceed by taking the rule corresponding to the constructor δ and applying it to the constructor's data p and the input value x. If our derivation tree is well-typed, this will result in a value of the action type σ of δ's rule (metalanguage non-termination is not possible). If the action type allows for immediate blocking, then the resulting value may have the form $\mathsf{in}(0, \sigma, \{\})$, which means that object language blocking will occur. Otherwise, there are two possibilities. The resulting value may have the form $\mathsf{in}(1, \sigma, y)$, which means that the rule has immediately produced the output value y, in which case our derivation tree must be turned into a complete concrete derivation tree with output y. On the other hand, the value may have the form $\mathsf{in}(i+2, \sigma, \{a, y, f\})$, which is a request to initiate a sub-evaluation. In this case, our derivation tree is turned into the incomplete concrete derivation tree

$$\mathsf{Inc}\langle \delta\, p, x, \langle \mathsf{Init}\langle a, y\rangle\rangle, f\rangle,$$

in which the sub-evaluation of term a with input y has been initiated, and the resumption f is waiting for the sub-derivation $\mathsf{Init}\langle a, y\rangle$ to become complete.

The next two tree expansion rules are similar, but are concerned with the expansion of incomplete concrete derivation trees whose last sub-derivations have become complete. Again, object language blocking is only possible if allowed by the action type σ of the metalanguage term that is being evaluated. Finally, the last rule is a contextual rule: it shows how the last sub-derivation of an incomplete concrete derivation tree can be expanded in place.

There is one more aspect to the semantics of Dops: the translation of concrete derivation trees into *abstract derivation trees.* Abstract derivation trees are defined as certain functions from tree paths to tree nodes consisting of either term/input pairs a, x or output values y. Then, a tree abstraction function abs can be defined in such a way that $\gamma \to \gamma'$ implies that $\mathsf{abs}\,\gamma \subseteq \mathsf{abs}\,\gamma'$. Resumptions are discarded as part of the abstraction process. Then, the meaning of a term/input pair a, x can be defined to be the abstract derivation tree

$$\bigcup\{\,\mathsf{abs}\,\gamma \mid \mathsf{Init}\langle a, x\rangle \to^* \gamma\,\}.$$

The meaning of a term/input pair will be a complete (and finite) derivation tree like the tree of Figure 4, or a blocked incomplete derivation tree like the one of Figure 5, or an infinite incomplete derivation tree like the one of Figure 6.

The implementation of Dops will allow users to construct as much of the meanings of term/input pairs as they desire. At each point in the evaluation of a given term/input pair, the user will be presented with a single node of the abstraction of the current concrete derivation tree, since the whole abstract derivation tree will typically be far too large to display in its entirety. The user will be able to navigate around the abstract derivation tree, and to view as much of the metalanguage values occurring in the tree's nodes as they wish (these values may themselves become too large to display fully). They may also opt to view the resumptions that occur in the underlying concrete derivation tree, which will normally be hidden. There will be various commands for continuing the tree expansion process, causing more of the final meaning to be constructed.

Dops specifications of the following operational semantics have been written: a typing system for the simply typed lambda calculus, substitution-based, big- and small-step semantics for the call-by-value untyped lambda-calculus, and big- and small-step semantics for a simple imperative language. When expressing a small-step semantics in Dops, one will make use of output types involving terms. It is also useful to add an extra layer to a small-step semantics that computes the transitive closure of the original small-step relation. Dops is currently being implemented in Standard ML.

3 Comparison with related work

There are various operational semantics (or logical) frameworks that allow users to evaluate object language term/input pairs [2,1,6,7,4]. Some of these frameworks support the construction of complete derivation trees in cases when term/input pairs evaluate to output values [1,6,4]. But, as far as I know, only D. Berry's Animator Generator [1] supports the incremental construction of derivation trees.

The Animator Generator takes in a deterministic operational semantics for

a programming language and generates an animator for that language. The animator incrementally constructs derivation trees, starting from term/input pairs, and displays various views of those trees. One of these views, which Berry calls a "semantic view", displays whole derivation trees. From our point of view, however, the Animator Generator suffers from several deficiencies.

The Animator Generator doesn't do a good job of displaying derivation trees (supporting other kinds of views had much higher priority). The main problem is that it insists on displaying an entire derivation tree in a single window; when the tree becomes large, this makes it very difficult to navigate around the tree. The problem is compounded by the fact that semantic values are also displayed in their entirety. We hope that our approach to displaying derivation trees will work better in practice.

In the Animator Generator's metalanguage, auxiliary tests and functions must be expressed as separate sets of rules. Unfortunately, this means that the side-conditions and auxiliary operations of rules may fail to be total (i.e., may diverge), which can lead to apparent rule blocking that won't be detected by the system. This deficiency is shared by all of the frameworks referred to above, but is avoided in Dops by employing a metalanguage in which all terms converge.

Finally, the tree expansion model of the Animator Generator is much more complicated than our definition of tree expansion (Figure 11) via a small-step semantics on concrete derivation trees.

The incremental construction of derivation trees has also been advocated by Gunter and Rémy [3]. They gave a definition of "partial proofs" in the context of a big-step semantics of a simple programming language. However, they only gave an informal description of how one partial proof can be transformed into another, by "resolving or extending subgoals" in the first.

4 Acknowledgments

Dave Schmidt's interest in viewing the semantics of divergent programs as infinite big-step derivation trees led to this work. (My suggestion to Dave that derivation trees be incrementally constructed using a small-step semantics on derivation trees contributed to the definition of the tree expansion relation in [5].) It is a pleasure to acknowledge stimulating conversations regarding this work with Dave, Brian Howard, Husain Ibraheem and Alan Jeffrey. Thanks are also due to Karen Bernstein, Bob Harper and Flemming Nielson for referring me to related research, and to Andy Pitts for helpful advice on terminology.

References

[1] D. Berry. *Generating Program Animators from Programming Language Semantics.* PhD thesis, Department of Computer Science, University of Edinburgh, 1991. Report number ECS-LFCS-91-163.

[2] P. Borras, D. Clément, T. Despeyroux, J. Incerpi, G. Kahn, B. Lang, and V. Pascual. CENTAUR: the system. In *ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 14–24. ACM Press, 1988.

[3] C. A. Gunter and D. Rémy. A proof-theoretic assessment of runtime type errors. Technical Report 11261-921230-43TM, AT&T Bell Laboratories, 1993.

[4] P. H. Hartel. LATOS—a Lightweight Animation Tool for Operational Semantics. Technical Report DSSE-TR-97-1, Declarative Systems and Software Engineering Group, Department of Electronics and Computer Science, University of Southampton, 1997.

[5] H. I. Ibraheem and D. A. Schmidt. Partial evaluation of higher-order natural-semantics derivations. In M. Leuschel, editor, *Workshop on Specialization of Declarative Programs and its Applications*, pages 17–28, Port Jefferson, Long Island, NY, 1997.

[6] S. Michaylov and F. Pfenning. Natural semantics and some of its meta-theory in Elf. In *Second International Workshop on Extensions of Logic Programming*, number 596 in Lecture Notes in Artificial Intelligence, pages 299–344, 1991.

[7] M. Pettersson. A compiler for natural semantics. In *Sixth Conference on Compiler Construction*, volume 1060 of *Lecture Notes in Computer Science*, pages 177–191. Springer-Verlag, 1996.

Computing with Contexts
A simple approach

David Sands

Department of Computing Science,
Chalmers University of Technology and Göteborg University,
S-412 96 Göteborg, Sweden; dave@cs.chalmers.se

Abstract

This article describes how the use of a higher-order syntax representation of contexts [due to A. Pitts] combines smoothly with higher-order syntax for evaluation rules, so that definitions can be extended to work over contexts. This provides "for free" — without the development of any new language-specific context calculi — evaluation rules for contexts which commute with hole-filling. We have found this to be a useful technique for directly reasoning about operational equivalence. A small illustration is given based on a unique fixed-point induction principle for a notion of guarded context in a functional language.

1 About contexts

The notion of a context is widely used in programming language semantics — for example in the definition of operational equivalences, or program transformation, and in certain styles of operational semantics definitions.

A context is just a term with some *holes*. The holes are place-holders for missing subterms. Each hole may occur zero or more times, and the process of *filling a hole with a term* is the textual operation of replacing all occurrences of a hole by the corresponding term. For most of this article we will consider contexts with just one hole, and that hole may occur zero or more times in the context.

The difference between hole-filling and the usual notion of substitution arises when the language contains binding operators; filling a hole with a term may cause variable in the term to be bound.

For example, the lambda-calculus context $(\lambda x.[\,])\lambda x.\lambda y.[\,]$ is a context with one hole, written $[\,]$, and this hole occurs twice. Call this context C. Filling the hole in C with the term $x\ y$, which is typically denoted by $C[x\ y]$, results in the term $(\lambda x.x\ y)\lambda x.\lambda y.x\ y$. Note that the resulting term has one free occurrence of the variable y, and that the variable x has been *captured* — in this example by two different lambda abstractions. Because of this possibility

of variable-capture (even in the case when a hole occurs just once in a term), such contexts cannot be identified up to renaming of bound variables, since renaming does not "commute with hole filling". In this example:

$$\begin{array}{ccc} (\lambda x.[\,])\lambda x.\lambda y.[\,] & \xrightarrow{\text{fill with } x\ y} & (\lambda x.x\ y)\lambda x.\lambda y.x\ y \\ \alpha\text{-convert}\downarrow & & \vdots \\ (\lambda z.[\,])\lambda x.\lambda y.[\,] & \xrightarrow{\text{fill with } x\ y} & (\lambda z.x\ y)\lambda x.\lambda y.x\ y \end{array}$$

Note that the terms of the right-hand side are not α-convertible. Such problems arise when one attempts to argue properties about the behaviour of "terms-in-context". For example, in direct proofs about contextual equivalence.

Representations of Contexts: Previous work

Talcott and Mason *et al* [MT91,AMST97,Tal97] present some direct proofs about contextual equivalence in a lambda calculus with effects. In order to be made rigorous, these arguments require the development of a calculus of generalised contexts. This development is based on Talcott's earlier work on a theory of binding structures [Tal92,Tal93]; related work is reported by Mason [Mas96].

The Talcott/Mason approach takes the following form. For the particular term language under study one must

(i) introduce a generalised definition of contexts where holes are decorated with (generalised) substitutions;

(ii) establish basic definitions for generalised contexts, such as substitution and hole-filling;

(iii) lift the definition of computation (reduction) up to generalised contexts

(iv) establish the main result: that reduction "commutes with hole filling"

One can motivate the Talcott/Mason representation of contexts by considering the last point: the problem of extending the definition of reduction to work on contexts in such a way that it commutes with hole-filling. Consider the lambda-calculus context $(\lambda x[\,])I$ (where I is the identity function, $\lambda x.x$). If one extended β-reduction naïvely to contexts we would obtain:

$$(\lambda x[\,])I \rightarrow_\beta [\,]$$

This is clearly not adequate, since e.g. filling the hole with x does not "commute" with this reduction:

$$\begin{array}{ccc} (\lambda x.[\,])I & \xrightarrow{\quad\beta\quad} & [\,] \\ \text{fill with } x\downarrow & & \downarrow\text{fill with } x \\ (\lambda x.x)I & \xrightarrow{\beta} I \quad \not\equiv & x \end{array}$$

The problem here is that the reduction step "forgets" the term I. The solution

is to decorate holes with explicit substitutions, so that e.g.,

$$(\lambda x[\,])I \to_\beta [\,]^{[I/x]}$$

But once this extension is made, the range of substitutions must also be permitted to contain generalised contexts, so that e.g.,

$$(\lambda y.[\,])[\,]^{[I/x]} \to_\beta [\,]^{[[\,]^\theta/y]} \qquad \text{where } \theta = [I/x]$$

Summary
In this note we show how an alternative approach to representing contexts significantly simplifies, and to some extent generalises the "context calculus" that is required to compute with contexts.

- The first simplification is that the representation of contexts — which is due to Pitts [Pit94] — is based on higher-order syntax (i.e. typed lambda-calculus as a syntactic meta-language), so no new calculus needs to be developed;
- The second simplification is that we show how many common definitions involving terms, e.g., evaluation relations, reduction, abstract-machine steps and sets of terms or contexts specified by grammars, can be "automatically" extended to contexts in such a way that they commute with hole-filling "for free".
- It generalises previous approaches in the sense that it is not tied to a particular syntax or a particular relation involving terms (i.e. reduction).

The author has already made extensive use of these techniques in a number of proofs about operational semantics; the initial motivations for this work were the proofs about the GDSOS operational semantics rule-format presented in [San97]. The proofs[1] of many of the results reported there build on the ideas presented in this note.

There are a few alternative context-calculi that have appeared in the literature. Lee and Friedman [LF96] propose a calculus in which contexts are regarded as concrete representations (source code) for terms. More recently, Hashimoto and Ohori [HO98] describe a context calculus which extends lambda calculus to include first-class contexts (via context abstraction and context application (= hole-filling). Their main result is that the calculus is confluent, and this is achieved with the help of a type system. Their calculus involves labelling hole variables with renamings, which is reminicent of Talcott's approach.

The remainder of this note is organised as follows: In section 2 we introduce Pitts' representation of contexts. In section 3 we show how this representation combines smoothly with the use of higher-order syntax in term-based definitions (e.g. operational semantics rules), so that definitions extend to

[1] Due to space limitations, these proofs do not appear in the conference article

contexts "for free". By way of illustration, in the concluding section we look at a small application of the ideas to the proof of a unique fixed-point theorem (in the style of guarded-recursion theorems in process calculus) for a lazy lambda-calculus with constructors.

2 A Second-order Representation of Contexts

The definition of contexts which we adopt here was introduced by Pitts in [Pit94]. Pitts' main motivation for adopting a non-standard definition of contexts appears to be that standard contexts cannot be identified up to α-equivalence.

Pitts solves this problem by using *function variables* to represent holes, and to represent hole filling by *substitution* of meta-abstractions for these function variables. This representation does indeed enable contexts to be identified up to renaming of bound variables; the key point of this note is that the representation allows hole-filling to "commute" with many other relations involving terms, not just α-equivalence.

The basic idea can be illustrated by some examples. A hole will be represented by an application of some function-variable ξ to a vector of variables. Each function variable has a given *arity*, which dictates exactly how many arguments it expects. For example, the conventional context $(\lambda x[\,])I$ could be represented by $(\lambda x.\xi(x))I$ (so in this case ξ has arity 1). This representation of the context can be α-converted in the usual way. Hole-filling is represented by substituting a *meta-abstraction* for ξ. Filling $(\lambda x[\,])I$ with x will now be represented by applying the substitution $[{}^{(x)x}/_{\xi}]$

$$\begin{aligned}((\lambda x.\xi(x))I)[{}^{(x)x}/_{\xi}] &\equiv ((\lambda y.\xi(y))I)[{}^{(x)x}/_{\xi}] \\ &\equiv ((\lambda y.(x)x \cdot (y))I) \\ &\equiv ((\lambda y.y)I)\end{aligned}$$

In the second line we have informally written a meta-syntactic application $(x)x \cdot (y)$ to represent the application of the abstraction $(x)x$ to the variable y is reduced to y. Since we only need second-order function-variables (i.e., function variables which range over abstractions of terms) these meta-beta-reductions can be incorporated into the definition of substitution itself.

Notice also with this example that we can now β-reduce the context:

$$((\lambda x.\xi(x))I) \rightarrow_{\beta} \xi(I)$$

and now we get $\xi(I)[{}^{(x)x}/_{\xi}] \equiv I$ as we hoped. As we shall see in the next section, the fact that this works boils down to a simple associativity property of substitutions (the standard lambda-calculus substitution lemma).

Term Syntax

In the general definitions which follow we will adopt a type-theory-style abstract syntax for terms (see e.g. [HL78,MN95,NPS90,PE88]). For specific examples we will use the familiar terms from the lambda-calculus, and their usual concrete syntax.

First we fix a countably infinite set *Var* of ordinary variables. A language L is specified by a set of operators O of a fixed arity. As usual, the arity specifies the number of *operands* for each operator, but it specifies more than just this, since we wish to specify the syntax of operators with binding. Each operand is possibly an abstraction, i.e., a list of zero or more distinct variables followed by a term, where the variables are considered bound in the term. The arity of an operator is therefore given by a sequence of natural numbers; the length of the sequence is the number of operands, and the natural numbers are the number of bound variables associated with the corresponding operand. For example, the terms of the lambda calculus would be represented in this syntax by the set of operators $\{\lambda, \boldsymbol{apply}\}$ with $\text{arity}(\lambda) = (1)$, $\text{arity}(\boldsymbol{apply}) = (0, 0)$.

Let x, y, etc., range over *Var*, and let $\boldsymbol{p}$, $\boldsymbol{q}$ range over O.

The terms of L, T, ranged over by M, N are defined inductively as follows:

$$\frac{}{x \in T} \qquad \frac{M_1 \in T \cdots M_n \in T}{\boldsymbol{p}((\bar{x}_1)M_1, \ldots, (\bar{x}_n)M_n) \in T} \quad \text{arity}(\boldsymbol{p}) = (k_1, \ldots k_n)$$

each $\bar{x}_i$ is a list of k_i distinct variables.

For example, the term $(\lambda x.y)z$ would be written in this abstract syntax as $\boldsymbol{apply}(\boldsymbol{\lambda}((x)y), z)$.

Contexts

Now we can be more precise about the definition of contexts. We follow [Pit94] quite closely, albeit with a more general term-syntax, (but a more casual treatment of free variables).

Contexts are an extension of terms to include hole-variables. Fix a countably infinite set *HVar* of hole variables. Each hole variable ξ, has an associated arity (which we will also denote $\text{arity}(\xi)$) which is a natural number. Hole variables of arity n will range over abstractions of the form: $(x_1, \ldots, x_n)M$.

The contexts $T^\star$, ranged over by $\mathbb{C}$, $\mathbb{D}$, $\mathbb{C}'$ etc are defined inductively as follows:

$$\frac{}{x \in T^\star} \qquad \frac{\mathbb{C}_1 \in T^\star \cdots \mathbb{C}_n \in T^\star}{\xi(\mathbb{C}_1, \ldots, \mathbb{C}_n) \in T^\star} \quad \text{arity}(\xi) = n$$

$$\frac{\mathbb{C}_1 \in T^\star \cdots \mathbb{C}_n \in T^\star}{\boldsymbol{p}((\bar{x}_1)\mathbb{C}_1, \ldots, (\bar{x}_n)\mathbb{C}_n) \in T^\star} \quad \text{arity}(\boldsymbol{p}) = (k_1, \ldots k_n)$$

each $\bar{x}_i$ is a list of k_i distinct variables.

Hole Filling

Hole filling is defined by substitution. The usual definition of substitution of terms for variables is routinely extended to substitution of contexts for variables. We use the notation $\mathbb{C}[\bar{\mathbb{C}}/\bar{x}]$ to denote the result of simultaneous substitution of contexts $\bar{\mathbb{C}} = \mathbb{C}_1, \ldots, \mathbb{C}_n$ for some distinct variables $\bar{x} = x_1, \ldots, x_n$.

Substitution in the case of hole variables requires a little more attention. If ξ is a hole variable of arity k, then we need to define the result of substituting a meta-abstraction of the form $(x_1, \ldots, x_k)\mathbb{D}$ for occurrences of ξ in a context $\mathbb{C}$.

The definition of substitution is much as one would expect, inductively following the term-structure, and avoiding free-variable capture along the way. The interesting case is the following:

$$\xi(\mathbb{C}_1, \ldots, \mathbb{C}_k)[(x_1, \ldots, x_k)\mathbb{D}/\xi] = \mathbb{D}[\bar{\mathbb{C}}'/\bar{x}]$$
$$\text{where } \bar{\mathbb{C}}' = \mathbb{C}_1[(x_1, \ldots, x_k)\mathbb{D}/\xi], \ldots, \mathbb{C}_k[(x_1, \ldots, x_k)\mathbb{D}/\xi]$$

This step can be (informally) broken down into two stages,

(i) the substitution of $(\bar{x})\mathbb{D}$ for ξ to yield

$$(\bar{x})\mathbb{D} \cdot (\mathbb{C}_1[(\bar{x})\mathbb{D}/\xi], \ldots, \mathbb{C}_k[(\bar{x})\mathbb{D}/\xi])$$

(ii) the meta-β-reduction of the application to give

$$\mathbb{D}[\mathbb{C}_1[(\bar{x})\mathbb{D}/\xi], \ldots, \mathbb{C}_k[(\bar{x})\mathbb{D}/\xi]/\bar{x}]$$

Of course this is only informal, since the meta-application which appears after step 1 is not part of the syntax.

About substitution

Something which should be borne in mind when considering the presentation of syntax that we are using is that it is really just a fragment of a simply typed lambda-calculus. There is just one base-type, o, representing the type of terms. An operator of arity e.g., $(0, 2)$ can be thought of as a constant of type $(o \times ((o \times o) \to o)) \to o$. An ordinary variable has type o, and a hole-variable of arity n can be thought of as having type

$$\underbrace{(o \times \cdots \times o)}_{\text{n}} \to o$$

Thinking of our syntax as a typed lambda calculus one should note that we only consider terms which are head-normal forms. One could add meta-application and projections to the syntax to obtain a more complete syntactic metalanguage (as in Martin-Löf's theory of arities [NPS90]) although we do not consider this necessary for present purposes.

It should now be no surprise that certain standard results from the lambda-calculus carry over to contexts. We mention one such result which will be

useful in the next section: a cut-down version of the *substitution lemma*, which states a commutativity property of substitutions:

Lemma 2.1 (Substitution Lemma) *If* $y \notin \text{FV}(\mathbb{D}) \setminus \bar{x}$ *then*

$$\mathbb{C}[(\bar{x})\mathbb{D}/\xi][N/y] \equiv \mathbb{C}[N/y][(\bar{x})\mathbb{D}/\xi]$$

Representing conventional contexts

If we are to use this alternative – and more general – representation of contexts in order to facilitate reasoning about e.g. contextual (operational) equivalence, then it is important to understand the connection to the conventional notion of context.

Conventional contexts correspond to a strict subset of contexts — namely those in which all occurrences of a hole variable ξ occur as $\xi(\bar{x})$ for some distinct variables $\bar{x}$.

For a conventional context C containing zero or more occurrences of a hole $[\,]$, let $\mathit{traps}(C)$ denote the set of the variables which are in scope at at least one occurrence of the hole in C – i.e. the set of variables which may become trapped (captured) at some occurrences when the hole is filled. So for example $\mathit{traps}((\lambda x.[\,])(\lambda y.[\,])) = \{x, y\}$. Let $\overline{\mathit{traps}}(C)$ denote some canonical vector of the trapped variables (e.g., listed from left-to-right according to the bindings in C).

For each context $\mathbb{C}$, we inductively define the mapping $\langle\cdot\rangle_{\mathbb{C}}$ which takes a conventional context to a generalised context by replacing holes with the context $\mathbb{C}$:

$$\langle x\rangle_{\mathbb{C}} = x$$

$$\langle \boldsymbol{p}(\,(\bar{x}_1)C_1, \ldots, (\bar{x}_n)C_n)\rangle_{\mathbb{C}} = \boldsymbol{p}(\,(\bar{x}_1)\langle C_1\rangle_{\mathbb{C}}, \ldots, (\bar{x}_n)\langle C_n\rangle_{\mathbb{C}})$$

$$\langle [\,]\rangle_{\mathbb{C}} = \mathbb{C}$$

In other words, mixing the conventional and general context notations we could say that $\langle C\rangle_{\mathbb{C}} = C[\mathbb{C}]$.

A conventional context C can be represented by $\langle C\rangle_{\xi(\bar{x})}$, where $\bar{x}$ are the captured variables of C. Then the operation of filling C with the term M is represented by the substitution $[(\bar{x})M/\xi]$. The following lemma makes this claim precise:

Lemma 2.2 *For any conventional context* C*, and any term* M*, if* $\overline{\mathit{traps}}(C) = \bar{x}$ *and* ξ *is any hole variable of arity* $|\bar{x}|$*, then*

$$C[M] \equiv \langle C\rangle_{\xi(\bar{x})}[(\bar{x})M/\xi]$$

The proof is by induction on the structure of C.

The extended definition of contexts (henceforth called simply contexts) subsume conventional contexts; conventional contexts correspond to contexts with one hole variable, and for which all occurrences are identically of the form $\xi(\bar{x})$ for some vector of distinct variables $\bar{x}$.

3 Extending Definitions from Terms to Contexts

The purpose of this section is to show how typical syntax-oriented definitions can be lifted to operate over contexts in a natural way.

We proceed by considering a tiny example: an evaluation relation for the lazy lambda calculus.

$$\frac{M \Downarrow \lambda x.M' \quad M'[N/x] \Downarrow N'}{MN \Downarrow N'}$$

$$\frac{}{\lambda x.M \Downarrow \lambda x.M}$$

We would like to lift this definition to contexts in the following obvious way:

$$\frac{\mathbb{C} \Downarrow \lambda x.\mathbb{C}' \quad \mathbb{C}'[\mathbb{D}/x] \Downarrow \mathbb{D}'}{\mathbb{C}\mathbb{D} \Downarrow \mathbb{D}'}$$

$$\frac{}{\lambda x.\mathbb{C} \Downarrow \lambda x.\mathbb{C}}$$

What is more, for this definition to be useful we would like to be sure that hole-filling and the evaluation relation commute. We could just knuckle down and prove this, but the point we wish to make in this section is that this will *always* work for syntax oriented definitions, by virtue of the representation of contexts. To see why this is so, we will need to be more formal about the rules which make up such inductive definitions.

Formal rules

In order give a precise meaning of rules such as those above we switch to the second-order syntax for terms and introduce a syntax for *meta terms.* For the terms of a given language, fix a countable set of *metavariables Mvar* ranged over by $\mathcal{X}$, $\mathcal{Y}$, etc. Metavariables will range over both terms and abstractions of terms, and will be used to formalise rules such as those above. Metavariables will be assumed disjoint from hole variables and ordinary variables. Just as for hole variables, with each metavariable $\mathcal{X}$, we associate an arity which is a natural number. The idea is that metavariables of arity 0 will range over terms, while meta variables of higher arity will range over abstractions. Value metavariables always have arity 0.

Definition 3.1 *To define the meta-terms for a given language we define an indexed set of meta-abstractions* $\{MT_i\}_{i \geq 0}$, *(ranged over by* $\mathcal{M}$, $\mathcal{N}$, *etc.). MT_0 are the meta-terms proper, used to denote terms in formal definitions. For each $k > 0$, MT_k is the set of meta-terms which represent k-variable abstractions of terms. The raw syntax of meta-terms follows the syntax of terms, with the exception of a meta-application operator, which appears in the form* $\mathcal{X} \cdot (\mathcal{M}_1, \ldots, \mathcal{M}_n)$. *These sets are given inductively by the following*

rules:

$$\frac{}{x \in MT_0} \qquad \frac{}{\mathcal{X} \in MT_n}\ \text{arity}(\mathcal{X}) = n$$

$$\frac{\mathcal{M} \in MT_0}{(x_1, \ldots, x_n)\mathcal{M} \in MT_n}$$

$$\frac{\mathcal{M}_1 \in MT_0 \cdots \mathcal{M}_n \in MT_0}{\mathcal{X} \cdot (\mathcal{M}_1, \ldots, \mathcal{M}_n) \in MT_0}\ \text{arity}(\mathcal{X}) = n$$

$$\frac{\mathcal{M}_1 \in MT_{k_1} \ldots \mathcal{M}_n \in MT_{k_n}}{\boldsymbol{p}(\mathcal{M}_1, \ldots, \mathcal{M}_n) \in MT_0}\ \text{arity}(\boldsymbol{p}) = (k_1 \ldots k_n)$$

One can easily see that meta-terms include the terms of the language. Meta-terms are used to formalise syntax-oriented definitions. The rules above can be formalised as:

$$\frac{\mathcal{X} \Downarrow \lambda\mathcal{X}_1 \quad \mathcal{X}_1 \cdot (\mathcal{Y}) \Downarrow \mathcal{Z}}{\boldsymbol{apply}(\mathcal{X}, \mathcal{Y}) \Downarrow \mathcal{Z}}$$

$$\frac{}{\lambda\mathcal{X}_1 \Downarrow \lambda\mathcal{X}_1}$$

A *raw instance* of a rule is obtained by applying a substitution to the metavariables in a rule. A substitution replaces metavariables by term-abstractions of the corresponding arity. For example, the substitution $\sigma = [{}^{(x)xx}/_{\mathcal{X}_1}]$ applied to the rule-schema (which incidentally has zero premises) $\lambda\mathcal{X}_1 \Downarrow \lambda\mathcal{X}_1$ gives $\lambda(x)xx \Downarrow \lambda(x)xx$ The *valid* instances (usually just called rule-instances) are defined inductively in the obvious way as a the raw instances for which each premise is a conclusions for some valid instance. There may be other side-conditions, e.g., that all instances involve only closed terms.

Extending Instances to Contexts

Any collection of rule schemas can thus be viewed as inductively generating certain sets. Lifting these definitions to contexts is now trivial: *simply allow the instances of a rule to contain hole variables.* In other words we allow substitution instances of a rule to replace metavariables by contexts (or abstractions of contexts). We will call such an instance a *context rule instance.*

Let us consider a concrete example. Given the formal rule for beta-reduction:

$$(\lambda\mathcal{X}_1)\mathcal{Y} \rightarrow_\beta \mathcal{X}_1 \cdot (\mathcal{Y})$$

where for the sake of readability we have written application in the usual implicit form, we can construct a rule instance by applying the substitution $[{}^{(x)\xi(x),\ \xi(x)}/_{\mathcal{X}_1,\ \mathcal{Y}}]$ which gives:

$$(\lambda(x)\xi(x))\ \xi(x) \rightarrow_\beta \xi(\xi(x))$$

Generalising "evaluation/reduction commutes with hole filling"

The generalisation of the idea that "evaluation/reduction commutes with hole filling" is that filling the holes in a valid rule-instance yields a valid rule instance.

Theorem 3.2 *Let $\frac{P}{c}$ denote a formal rule with a set of premises P and a conclusion c. Let σ be some substitution of terms for metavariables such that $(\frac{P}{c})\sigma$ is a generalised instance of the rule. Now suppose that τ is a hole-filling (a substitution of hole-variables for term abstractions of the corresponding arity). Then the following are identical rule-instances:*

$$\left(\left(\frac{P}{c}\right)\tau\right)\sigma \equiv \left(\frac{P}{c}\right)(\sigma\tau)$$

where $(\sigma\tau)$ denotes the application of substitution τ to the range of σ.

PROOF. It is easy to see that valid rule-instances are closed under substitution, and hence that $((\frac{P}{c})\sigma)\tau$ is a valid rule instance. Since metavariables and hole variables are distinct, then the equivalence above follows immediately from the substitution lemma. □

Other Syntactic Categories

The idea of extending definitions to work over contexts is widely applicable. Definitions in operational semantics often involve the construction of several syntactic categories which either contain terms or restrict the set of terms in some way. For example,

- The definition of *configurations* in an SOS-style semantics or in the definition of an abstract machine containing e.g. *stacks* of terms or *environments* (finite mappings from variables to terms). The definition of a rule-instance is essentially the same, and so there are no problems in allowing instances to contain hole-variables.
- The definition of particular subsets of terms, e.g., the *values* in a call-by-value functional language $V ::= \mathit{constant} \mid \langle V_1, V_2\rangle \mid \lambda x.M \mid \ldots$ can be lifted to "value contexts" in the obvious way; value metavariables used in computation rules must then be instantiated with value contexts.
- A popular style of small-step operational semantics is to use *evaluation contexts* to describe a deterministic reduction strategy. An evaluation context, usually specified by a grammar, is a context with exactly one hole. For example, the evaluation contexts for a call-by-value lambda calculus with strict pairing and left-to-right evaluation might be specified by

  $$E ::= [\,] \mid E\,M \mid V\,E \mid \langle E, M\rangle \mid \langle V, E\rangle \mid \ldots$$

  The evaluation rules can then be specified by e.g.

  $$E[(\lambda x M)V] \rightarrow E[M[{}^{V}/_{x}]]$$

  Formalising this style of definition presents no problems either: evaluation contexts (for this language at least) are just abstractions of one variable

(the hole). Note that in this particular example there are three definitions which need to be generalised: values, evaluation contexts and the evaluation rules themselves. The only precaution is to treat the hole variable in the definition of evaluation contexts (which in this example can be taken to have arity zero) as being distinct from all other hole variables.

4 Applications

A typical application of "direct reasoning about contexts" is to prove properties about a contextually-defined equivalence relation. Examples of this style of "direct" reasoning can be found in e.g., [MT91,AMST97]; using the approach described here we can make such arguments rigorous with almost no overhead in building a language-specific context calculus. We used this technique for several of the results described in [San97], where various theorems about operational preorders are established for any language whose operational semantics rules fit a certain *rule format.*

In [MS98,Mor98] the approach described in this article is used to establish a *context lemma* for call-by-need lambda calculi (in the latter including a form of fair nondeterminism); in these applications the semantics is based on an abstract machine, rather than a term-based computation model.

Most applications of "context evaluation" build upon a small-step operational semantics of some kind. This is natural since the larger the computation step which is used, the less likely that the computation step can be applied to a context. In the remainder of this article we consider an example application where a large-step semantics still yields some useful computations on contexts.

4.1 A Unique Fixed-Point Induction Theorem

A well-known proof technique in e.g. process algebra involves syntactically characterising a class of recursion equations which have a unique solution. Knowing that a recursive definition has a unique fixed point means that one can prove equivalence with a recursively defined entity by showing that the recursion equation is satisfied.

The usual syntactic characterisation is that *guarded recursion*: if recursive calls are syntactically "guarded" by an observable action then the fixed-point of the definition is unique.

We illustrate related notion of *guarded context* for a lazy lambda-calculus with constants, and show – with the help of context evaluation – that guarded contexts enjoy a unique fixed-point property. The process calculus notions of guardedness are something of a panacea when it comes to reasoning about recursion. Although the functional notion of guardedness falls a long way short of this, there are still some interesting instances.

We will take an extension of the lazy lambda calculus [Abr90]: The syntax

of expressions (M, N etc) is as follows:

$$\begin{array}{lll} M ::= x \mid M\,N \mid \lambda x.M & & \text{(Var; Apply; Lambda)} \\ \quad \mid \textbf{case } L \textbf{ of } \{\mathbf{c}_1(\bar{x}_1) \Rightarrow M_1 \ldots \mathbf{c}_n(\bar{x}_n) \Rightarrow M_n\} & & \text{(Case)} \\ \quad \mid \mathbf{c}(\bar{M}) & & \text{(Constructors)} \end{array}$$

Constructors have a fixed arity (≥ 0), and we implicitly assume that instances of constructor expressions and patterns always respect the arity. We will use a deductive ("big-step") style of semantics;

The results of computations are *weak head-normal forms* (WHNF): either constructor terms $\mathbf{c}(\bar{L})$ or lambda-abstractions $\lambda x.M$. Let V, W range over WHNF's. Now we define the *convergence* relation between closed terms by the evaluation rules in figure 1. As usual we write $M\Downarrow$ to mean $\exists N.\, M\Downarrow N$ Let

$$\frac{}{\lambda x.M \Downarrow \lambda x.M} \qquad \frac{}{\mathbf{c}(\bar{M}) \Downarrow \mathbf{c}(\bar{M})} \qquad \frac{M\Downarrow\lambda x.M \quad M[M_2/x]\Downarrow V}{M_1\ M_2 \Downarrow V}$$

$$\frac{L\Downarrow \mathbf{c_i}(\bar{L}) \quad M_i[\bar{L}/\bar{x}_i]\Downarrow V}{\textbf{case } L \textbf{ of } \{\cdots \mathbf{c_i}(\bar{x}_i) \Rightarrow M_i \cdots\} \Downarrow V}$$

Fig. 1. Convergence relation

$\sqsubseteq$ denote the operational preordering defined by $M \sqsubseteq N$ if and only if for all conventional contexts C such that $C[M]$ and $C[N]$ are closed expressions, then $C[M]\Downarrow$ implies $C[N]\Downarrow$. Let $\cong$ denote the corresponding equivalence relation.

Now we are in a position to define the guarded contexts:

Definition 4.1 *The* guarded contexts, $\mathbb{G}$, *are contexts containing at most one hole variable, and given by the following grammar:*

$$\mathbb{G} ::= M \mid \mathbf{c}(\mathbb{H}_1, \ldots, \mathbb{H}_n) \mid \lambda x.\mathbb{H} \mid \textbf{case } L \textbf{ of } \mathbf{c_1}(\bar{x}_1) \Rightarrow \mathbb{G}_1 \ldots \mathbf{c_n}(\bar{x}_n) \Rightarrow \mathbb{G}_n$$

$$\mathbb{H} ::= \xi(\bar{M}) \mid \mathbb{G}$$

Since guarded contexts have just one hole variable, we will write $\mathbb{G}[(\bar{x})M]$ to denote $\mathbb{G}[(\bar{x})M/\xi]$.

Theorem 4.2 (Unique Fixed Point) *For all expressions M, N where* FV*$(M)\cup$* FV*$(N) \subseteq \bar{x}$, the following proof rule is valid:*

$$\frac{M \cong \mathbb{G}[(\bar{x})M] \qquad N \cong \mathbb{G}[(\bar{x})N]}{M \cong N}$$

Using the justification given in the previous section, we tacitly extend the syntactic categories and definitions of one-step reduction and of convergence to allow the occurrence of hole variables. We will let $\mathbb{E}$ range over evaluation

contexts containing hole variables; $\mathbb{V}$ and $\mathbb{W}$ over weak head normal forms with holes, and we extend the definitions of one-step reduction and of convergence to these extended syntactic categories.

The main point of the example is that the proof of the above theorem is facilitated by the following property of guarded contexts:

Lemma 4.3 *For all guarded contexts $\mathbb{G}$ and all abstractions $(\bar{x})M$ such that $\mathbb{G}[(\bar{x})M]$ is closed,*

$$\exists V.\, \mathbb{G}[(\bar{x})M]\Downarrow V \iff \exists \mathbb{V}.\, \mathbb{G}\Downarrow \mathbb{V}$$

where $\mathbb{V}$ is a guarded context (i.e. either of the form $\lambda x.\mathbb{H}$ or $\mathbf{c}(\mathbb{H}_1, \ldots, \mathbb{H}_n)$

Proof. The ($\Leftarrow$) direction follows easily from Theorem 3.2, since it follows from $\mathbb{G}\Downarrow\mathbb{V}$ that $\mathbb{G}[(\bar{x})M]\Downarrow\mathbb{V}[(\bar{x})M]$. For the ($\Rightarrow$) direction, we assume that $\mathbb{G}[(\bar{x})M]\Downarrow V$ and proceed by rule induction to show that $\exists \mathbb{V}.\, \mathbb{G}\Downarrow\mathbb{V}$. We proceed by cases according to the structure of $\mathbb{G}$.

Case $\mathbb{G} \equiv M$: then $\mathbb{V}$ is simply V.

Case $\mathbb{G} \equiv \lambda x.\mathbb{H}$: then $\mathbb{V}$ is just $\lambda x.\mathbb{H}$. The case for constructors follows similarly.

Case $\mathbb{G} \equiv$ case L of $\{\mathbf{c_1}(\bar{x}_1) \Rightarrow \mathbb{G}_1 \cdots \mathbf{c_n}(\bar{x}_n) \Rightarrow \mathbb{G}_n\}$: then the rule for case evaluation provides the following inductive hypotheses: $L\Downarrow\mathbf{c_i}(\bar{L})$ for some $\mathbf{c_i}$, and $\mathbb{G}_i[\bar{L}/\bar{x}_i]\Downarrow\mathbb{V}$ for a guarded value context $\mathbb{V}$. The latter case relies on the observation that guarded contexts are closed under substitution. From the case evaluation rule we conclude that $\mathbb{G}\Downarrow\mathbb{V}$.

□

We sketch how the proof of the can be completed using the technique of "simulation up to" (for this particular language this technique is described in [San96], and is a simple adaptation of the (bi)simulation-style proof method).

Definition 4.4 (Simulation up to $\sqsubseteq$) *A relation $\mathcal{R}$ is a* simulation up to $\underset{\sim}{\sqsubseteq}$ *if for all M, N, whenever $M\ \mathcal{R}\ N$, then for all closing substitutions σ, if $M\sigma\Downarrow V$ then $N\sigma\Downarrow W$ for some W such that one of the following two conditions hold:*

(i) $V \equiv c(M_1 \ldots M_n)$, $W \equiv c(N_1 \ldots N_n)$ *and* $M_i \underset{\sim}{\sqsubseteq}; \mathcal{R}; \underset{\sim}{\sqsubseteq} N_i, i \in 1 \ldots n$

(ii) $V \equiv \lambda x.M'$ *and* $W \equiv \lambda x.N'$ *and* $M' \underset{\sim}{\sqsubseteq}; \mathcal{R}; \underset{\sim}{\sqsubseteq} N'$.

Proposition 4.5 *If $\mathcal{R}$ is a simulation up to $\underset{\sim}{\sqsubseteq}$ then $\mathcal{R} \subseteq \underset{\sim}{\sqsubseteq}$*

Now we can complete the proof of the theorem by constructing a suitable relation and showing that it is a simulation up to $\underset{\sim}{\sqsubseteq}$ (equivalence follows by symmetry of the argument).

Suppose that $M \cong \mathbb{G}_0[(\bar{x})M]$ and $N \cong \mathbb{G}_0[(\bar{x})N]$. We can assume without loss of generality that $\text{FV}(\mathbb{G}_0) \subseteq \bar{x}$. We will show that

$$R \stackrel{\text{def}}{=} \{\mathbb{G}[(\bar{x})M], \mathbb{G}[(\bar{x})N] \mid \text{FV}(\mathbb{G}) \subseteq \bar{x}\}$$

is a simulation up to $\underset{\sim}{\sqsubseteq}$. Suppose that $\mathbb{G}[(\bar{x})M]\sigma\Downarrow$. Since $(\bar{x})M$ is a closed

abstraction, $\mathbb{G}[(\bar{x})M]\sigma \equiv \mathbb{G}\sigma[(\bar{x})M]$. Since $\mathbb{G}\sigma$ is also a guarded context by lemma 4.3 we have that either

(i) $\mathbb{G}\sigma\Downarrow\lambda y.\mathbb{H}_0$ for some $\mathbb{H}_0$, or

(ii) $\mathbb{G}\sigma\Downarrow\mathbf{c}(\mathbb{H}_1 \ldots \mathbb{H}_n)$ for some constructor $\mathbf{c}$ and some $\mathbb{H}_1 \ldots \mathbb{H}_n$.

By theorem 3.2 we know that $\mathbb{G}[(\bar{x})N]\sigma\Downarrow$, and it remains to show that, in each respective case that $\mathbb{H}_i[(\bar{x})M] \underset{\sim}{\sqsubseteq} ; R ; \underset{\sim}{\sqsubseteq} \mathbb{H}_i[(\bar{x})N]$. By definition each $\mathbb{H}_i$ is either a guarded context or a hole. In the former case we have immediately that $\mathbb{H}_i[(\bar{x})M]\ R\ \mathbb{H}_i[(\bar{x})N]$ and so we are done by reflexivity of $\underset{\sim}{\sqsubseteq}$. In the latter case $\mathbb{H}_i$ is of the form $\xi(\bar{L})$ so we have that

$$\xi(\bar{L})[(\bar{x})M] \equiv M[\bar{L}/\bar{x}] \cong \mathbb{G}_0[(\bar{x})M][\bar{L}/\bar{x}]\ R\ \mathbb{G}_0[(\bar{x})N][\bar{L}/\bar{x}] \cong N[\bar{L}/\bar{x}]$$

as required.

Example

We leave the following example as an exercise which is easily proved using the unique fixed-point rule:

$$\texttt{map f (iterate f x)} \cong \texttt{iterate f (f x)}$$

where `map :: (a -> b) -> [a] -> [b]` and `iterate :: (a -> a) -> a -> [a]` are the usual Haskell recursive functions.

Acknowledgements

Thanks to Andy Moran for many helpful comments and discussions, and to Søren Lassen for suggestions for improvements to an earlier draft.

References

[Abr90] S. Abramsky. The lazy lambda calculus. In D. Turner, editor, *Research Topics in Functional Programming*, pages 65–116. Addison Wesley, 1990.

[AMST97] Gul A. Agha, Ian A. Mason, Scott F. Smith, and Carolyn L. Talcott. A foundation for actor computation. *Journal of Functional Programming*, 7(1):1–72, January 1997.

[HL78] G. Huet and B. Lang. Proving and applying program transformations expressed with second order patterns. *Acta Inf.*, 11(1):31–55, January 1978.

[HO98] M. Hashimoto and A. Ohori. A typed context calculus. Technical Report RIMS-1098, Research Institute for Mathematical Sciences, Kyoto University, 1998.

[LF96] S. Lee and D. Friedman. Enriching the lambda calculus with contexts: Toward a theory of incremental program construction. In *Proceedings of the 1996 ACM SIGPLAN International Conference on Functional Programming*, pages 239–250, May 1996.

[Mas96] I. Mason. Parametric computation. In *CATS'96*, 1996.

[MN95] R. Mayr and T. Nipkow. Higher-order rewrite systems and their confluence. Technical Report Sep26-1, Technical University of Munich, September 1995.

[Mor98] A. Moran. *Call-by-name, Call-by-need, and McCarthy's Amb.* PhD thesis, Department of Computing Sciences, Chalmers, Sweden, September 1998.

[MS98] A. Moran and D. Sands. A context lemma for call-by-need. Working Note. Jan 1998. Revised May, 1998.

[MT91] I. Mason and C. Talcott. Equivalence in functional languages with effects. *Journal of Functional Programming*, 1(3):287–327, July 1991.

[NPS90] Bengt Nordström, Kent Petersson, and Jan M. Smith. *Programming in Martin-Löf's Type Theory: An Introduction*, volume 7 of *International Series of Monographs on Computer Science.* Oxford University Press, 1990.

[PE88] F. Pfenning and C. Elliott. Higher-order abstract syntax. In *Proceedings of the SIGPLAN '88 Conference on Programming Lanugage Design and Implementation (SIGPLAN '88)*, pages 199–208. ACM Press, June 1988.

[Pit94] Andrew M. Pitts. Some notes on inductive and co-inductive techniques in the semantics of functional programs. Notes Series BRICS-NS-94-5, BRICS, Department of Computer Science, University of Aarhus, December 1994.

[San95] D. Sands. Total correctness by local improvement in program transformation. In *Conference Record of POPL '95: 22nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, New York, January 1995. ACM Press.

[San96] D. Sands. Total correctness by local improvement in the transformation of functional programs. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 18(2):175–234, March 1996. Extended version of [San95].

[San97] D. Sands. From sos rules to proof principles: An operational metatheory for functional languages. In *24th ACM SIGPLAN-SIGACT Symposium on Principles on Programming Languages (POPL'97).* ACM Press, 1997.

[Tal92] C. Talcott. Towards a theory of binding structures: An abstract algebra. In Maurice Nivat, Charles Rattray, Teodor Rus, and Giuseppe Scollo, editors, *Proceedings of the Second International Conference on Algebraic Methodology and Software Technology*, Workshops in Computing, pages 201–215, London, May22–25 1992. Springer Verlag.

[Tal93] C. Talcott. A theory of binding structures and applications to rewriting. *Theoretical Computer Science*, 112(1):99–143, April 1993.

[Tal97] C. Talcott. Reasoning about functions with effects. In A. Gordon and A. Pitts, editors, *Higher-Order Operational Techniques in Semantics.* Cambridge University Press, 1998.

Electronic Notes in Theoretical Computer Science 10 (1998)
URL: http://www.elsevier.nl/locate/entcs/volume10.html 20 pages

Flow Logic and Operational Semantics

Flemming Nielson

Computer Science Department
University of Aarhus
DK-8000 Aarhus C, Denmark

Hanne Riis Nielson

Computer Science Department
University of Aarhus
DK-8000 Aarhus C, Denmark

Abstract

Flow logic is a "fast prototyping" approach to program analysis that shows great promise of being able to deal with a wide variety of languages and calculi for computation. However, seemingly innocent choices in the flow logic as well as in the operational semantics may inhibit proving the analysis correct. Our main conclusion is that environment based semantics is more flexible than either substitution based semantics or semantics making use of structural congruences (like alpha-renaming).

1 Introduction

Flow logic facilitates the specification of program analyses [10] that automatically predict properties of programs holding in all executions. It allows to deal with a wide variety of languages; examples include the lambda-calculus with side-effects (a fragment of Standard ML) or communications (a fragment of Concurrent ML), several object based calculi, and a process algebra (the π-calculus). Analyses may be described in a *succinct* form (akin to program logic) or in a more *verbose* form (taking the form of constraint satisfaction); also they may be described at an *abstract* level of reasoning (using coinductive techniques) or in a more *compositional* manner (using inductive techniques). This allows to use the approach to first sketch the analysis, next refine it and prove it correct, and finally obtain an efficient implementation; furthermore, the development may be firmly rooted upon existing program analysis technology and insights, rather than having to start from scratch.

Structural operational semantics similarly allows to deal with a wide variety of languages. There are many choices that needs to be made concerning

how to define the semantics: e.g. having small-step or big-step transitions, using environments or performing direct substitution, making use of evaluation contexts or having explicit rules for reduction in context. Many of these choices are seemingly innocent in the sense that they do not affect the "meaning" of the language being defined; and indeed different formulations of the semantics can often be proved equivalent (although the proofs are sometimes quite laborious).

This might suggest that one could deal with a new language or calculus in the following way: first the syntax and informal meaning is defined, then the program analysis is developed simultaneously with the operational semantics, and finally they are consolidated with respect to one another (and in particular the analysis is proved correct). One advantage of this approach is that the fine details of the language definition are consolidated not only by semantic considerations but also by more pragmatic considerations concerning the ease with which programs can be validated not to have anomalous behaviour; we believe that this is a key issue in designing languages that are both theoretically well-behaved and pragmatically useful. Another advantage is that the methods would then be more likely to scale up to "real life" languages because different teams of researchers could be responsible for different aspects of the development; this is a major parameter for the success of formal methods in software engineering and is often neglected in purely theoretical studies.

In our experience the above approach is fraught with problems. One reason is that the structure, size and complexity of the correctness proofs depend on characteristica of the flow logic (e.g. whether it is abstract or compositional) as well as on characteristica of the operational semantics (e.g. whether it uses environments or direct substitution). In some cases the choices may "contradict" one another so that no proof of correctness is possible and the analysis or semantics has to be changed. It is therefore important to identify general guidelines concerning what complications are likely to arise for what combinations – in order that the use of formal methods in this area may become a craft rather than a (black) art.

Our main conclusion is that environment based semantics is more flexible than either substitution based semantics or semantics making use of structural congruences (like alpha-renaming) in terms of being able to accomodate a variety of specification styles for program analysis.

2 Setting the Scene

For simplicity this paper concentrates on an untyped lambda-calculus although analogous considerations apply to the more advanced object based and concurrent calculi mentioned above. The pure untyped lambda-calculus has the following syntax:

$e \in \mathbf{Exp}$
$e ::= x \mid \texttt{fn}\ x\ \texttt{=>}\ e \mid e\ e$

$x \in \mathbf{Var}$
where **Var** is an unspecified countably infinite set

The *succinct* formulations of flow logic considered here do not require that all program points are made explicit; so there is no need to place explicit labels on all subexpressions (as in [9]) or to convert programs into "A-normal-form" (as in [4]). Instead we shall *assume* that all function abstractions have initially been alpha-renamed so as to have distinct formal parameters that are also disjoint from the set of global variables, $\mathsf{FV}(e_*)$, of the program of interest, e_*.

Often program analysis is formulated as the demand to compute "the best" (or least) analysis information, $\hat{\rho}$, that pertains to a program, e_*: $\hat{\rho} = \mathsf{A}(e_*)$. Here we shall take the more flexible approach that a piece of analysis information, $\hat{\rho}$, needs to be validated with respect to the program, e_*: $\hat{\rho} \models e_*$ (yielding TT or FF). On the one hand this allows to develop algorithms for computing "the best" analysis information [11,5] and on the other hand it offers promise of analysing not only closed systems: whenever new expressions emerge from the environment it can be checked whether or not the current analysis has duly recorded all the possible effects of these expressions. The ability to analyse open systems is particularly important for calculi and languages dealing with distribution and mobility of software.

Example 2.1 To give an example of the analysis consider the following simple program, e_*, (in a slight extension of the syntax):

```
letrec g = (fn x => (g g))
in g (fn y => y)
```

Here a function g is defined that ignores its parameter and calls itself recursively upon itself; the function is then called with the identity function as parameter.

We shall next consider an analysis that is called a control flow analysis [15,16], a closure analysis [12,14] or a set based analysis [6]. To do so we first define the abstract environment $\hat{\rho}$ by:

$\hat{\rho}(\mathtt{g}) = \{\mathtt{fn\ x => (g\ g)}\}$
$\hat{\rho}(\mathtt{x}) = \{\mathtt{fn\ x => (g\ g)}, \mathtt{fn\ y => y}\}$
$\hat{\rho}(\mathtt{y}) = \emptyset$

The analysis of the program, e_*, then amounts to a judgement of the form

$\hat{\rho} \models e_* : \emptyset$

saying that it will be correct to stipulate that g is bound to the recursive function itself, that the formal parameter x is bound to the recursive function or the identity function, that the formal parameter y is bound to nothing (corresponding to the identity function never being called), and that the program never returns any function (corresponding to the fact that it loops forever). To validate the analysis we will encounter other judgements like

$\hat{\rho} \models \mathtt{fn\ x => (g\ g)} : \{\mathtt{fn\ x => (g\ g)}\}$

$\hat{\rho} \models \texttt{fn y => y} : \{\texttt{fn y => y}\}$
$\hat{\rho} \models \texttt{g g} : \emptyset$
$\hat{\rho} \models \texttt{g (fn y => y)} : \emptyset$

and to make this precise we need to clarify how the judgements, $\hat{\rho} \models e : W$, are defined; this will be the subject of Section 4 after having defined the operational semantics in Section 3.

3 Operational Semantics

Let us now start on our first task: the definition of the operational semantics. As already indicated there are a number of choices to be made. Here we shall just consider three kinds of semantics: a *substitution based* semantics in the manner of the λ-calculus [1], a structural operational semantics with *environments* in the manner of [13] and a variant involving *explicit alpha-renaming* of bound variables (in the manner of the structural congruence used in process algebras like the π-calculus).

All of these semantics are small-step and this is advantageous for the ability to express the correctness of looping programs and for programs with concurrency. The general form of the correctness result then is a subject reduction result:

if the analysis $\hat{\rho}$ is acceptable for the expression e_1
and if e_1 evolves into e_2
then the analysis $\hat{\rho}$ is acceptable for the expression e_2.

If a big-step semantics had been used then looping programs[1] as well as concurrent programs would present obstacles to the development.

Substitution based semantics
Perhaps the simplest kind of operational semantics uses substitutions rather than environments. In this case the operational semantics is given by a judgement of the form

$$e_1 \xrightarrow{s} e_2$$

saying that one step of evaluation of e_1 yields e_2. To define the judgement it is helpful to clarify that the expressions playing the role of values (i.e. fully evaluated expressions) are simply those function abstractions that only contain global variables:

$v \in \mathbf{Val}$
$v ::= \texttt{fn}\ x\ \texttt{=>}\ e$ provided that $\mathsf{FV}(\texttt{fn}\ x\ \texttt{=>}\ e) \subseteq \mathsf{FV}(e_*)$

Here the set $\mathsf{FV}(e)$ of free variables of an expression e is defined in the standard way: $\mathsf{FV}(x) = \{x\}$, $\mathsf{FV}(\texttt{fn}\ x\ \texttt{=>}\ e) = \mathsf{FV}(e) \setminus \{x\}$ and $\mathsf{FV}(e_1\ e_2) = \mathsf{FV}(e_1) \cup \mathsf{FV}(e_2)$.

[1] Assuming a standard inductive interpretation of big-step semantics.

The semantics is then defined by the following standard axioms and inference rules; intuitively we shall only be interested in evaluating upon expressions whose only free variables are among the global ones, but formally it suffices that this condition holds for the values:

$$\frac{e_1 \xrightarrow{s} e_1'}{e_1\ e_2 \xrightarrow{s} e_1'\ e_2}$$

$$\frac{e_2 \xrightarrow{s} e_2'}{v_1\ e_2 \xrightarrow{s} v_1\ e_2'}$$

$$v_1\ v_2 \xrightarrow{s} e[x \mapsto v_2] \text{ if } v_1 = (\texttt{fn}\ x\ \texttt{=>}\ e)$$

Here $e[x \mapsto v]$ denotes the expression e with all free occurrences of the variable x replaced by the value v. Since the formal parameters were assumed to be distinct from the global variables, $\mathsf{FV}(e_*)$, no variable capture can take place; hence there is no need for alpha-renaming any formal parameter and therefore the formal parameters continue to be distinct from the global variables. As usual these axioms and rules are to be interpreted inductively, meaning that the judgement is defined by what can be obtained using the axioms and rules and nothing else.

Environment based semantics

A somewhat more complex semantics is obtained by using environments instead of substitutions. Following the pattern in [13] we need to *extend the syntax* of the language in order to define the structural operational semantics. The changes needed for the extended syntax are as follows:

$ie \in \mathbf{IExp}$

$ie ::= x \mid \texttt{fn}\ x\ \texttt{=>}\ e \mid ie\ ie \mid \texttt{close}\ (\texttt{fn}\ x\ \texttt{=>}\ e)\ \texttt{in}\ \rho \mid \texttt{bind}\ \rho\ \texttt{in}\ ie$

$\rho \in \mathbf{Env}$

$\rho ::= [\,] \mid \rho[x \mapsto v]$

$v \in \mathbf{IVal}$

$v ::= \texttt{close}\ (\texttt{fn}\ x\ \texttt{=>}\ e)\ \texttt{in}\ \rho$

The expression $\texttt{close}\ (\texttt{fn}\ x\ \texttt{=>}\ e)\ \texttt{in}\ \rho$ encapsulates an unevaluated abstraction $\texttt{fn}\ x\ \texttt{=>}\ e$ with an environment ρ that gives values to all the free variables in $\texttt{fn}\ x\ \texttt{=>}\ e$; this construct is needed because we have decided to design a semantics using environments. The expression $\texttt{bind}\ \rho\ \texttt{in}\ ie$ encapsulates a partly evaluated expression ie with a local environment ρ; this construct is needed because we have decided to design a small-step semantics. Note that we retain the distinction between unevaluated expressions, $e \in \mathbf{Exp}$, and partly evaluated (or intermediate) expressions, $ie \in \mathbf{IExp}$, in order to clarify the precise points where partly evaluated expressions may occur; this will prove helpful when reasoning about the analysis. Finally, (intermediate) values are just closures, and environments are lists of mappings from variables to values: we write $\rho[x \mapsto v]$ for the environment ρ augmented such that the variable x

now maps to the value v; if there is more than one binding for a given variable we always use the rightmost (most recent) one.

The operational semantics is given by a judgement of the form:

$$\rho \vdash ie_1 \xrightarrow{E} ie_2$$

Here ρ is the environment in which the expression is to be evaluated. It is defined by the following standard axioms and inference rules:

$$\rho \vdash x \xrightarrow{E} v \text{ if } \rho(x) = v$$

$$\rho \vdash (\texttt{fn } x \texttt{ => } e) \xrightarrow{E} (\texttt{close } (\texttt{fn } x \texttt{ => } e) \texttt{ in } \rho)$$

$$\frac{\rho \vdash ie_1 \xrightarrow{E} ie_1'}{\rho \vdash ie_1\ ie_2 \xrightarrow{E} ie_1'\ ie_2}$$

$$\frac{\rho \vdash ie_2 \xrightarrow{E} ie_2'}{\rho \vdash v_1\ ie_2 \xrightarrow{E} v_1\ ie_2'}$$

$$\rho \vdash v_1\ v_2 \xrightarrow{E} (\texttt{bind } \rho_1[x \mapsto v_2] \texttt{ in } e) \text{ if } v_1 = (\texttt{close } (\texttt{fn } x \texttt{ => } e) \texttt{ in } \rho_1)$$

$$\frac{\rho_1 \vdash ie_1 \xrightarrow{E} ie_1'}{\rho \vdash (\texttt{bind } \rho_1 \texttt{ in } ie_1) \xrightarrow{E} (\texttt{bind } \rho_1 \texttt{ in } ie_1')}$$

$$\rho \vdash (\texttt{bind } \rho_1 \texttt{ in } v_1) \xrightarrow{E} v_1$$

Once again this definition is to be interpreted inductively.

Explicit alpha-renaming

Let us finally consider a variation of the environment-based semantics where there is an explicit rule for alpha-renaming the bound variable of an abstraction. This will allow us to illustrate some of the difficulties that will emerge when analysing process calculi like the π-calculus where a structural congruence (containing alpha-renaming) is defined and incorporated into the semantics.

The operational semantics is then given by a judgement of the form:

$$\rho \vdash ie_1 \xrightarrow{E\alpha} ie_2$$

As above ρ is the environment in which the expression is to be evaluated. The inductive definition is given by

analogues of the axioms and rules given for $\xrightarrow{E}$

together with the rule

$$\frac{ie =_\alpha ie' \quad \rho \vdash ie' \xrightarrow{E\alpha} ie'' \quad ie'' =_\alpha ie'''}{\rho \vdash ie \xrightarrow{E\alpha} ie'''}$$

where $ie =_\alpha ie'$ denotes that ie is equivalent to ie' modulo alpha-renaming (of formal parameters).

A similar modification, $\xrightarrow{S\alpha}$, is possible for the substitution based semantics; we shall write $\xrightarrow{\alpha}$ when it is not of any importance whether we refer to $\xrightarrow{E\alpha}$ or $\xrightarrow{S\alpha}$.

4 Flow Logic

Let us now turn to our second task: the specification of the program analysis. As already indicated there are a number of choices to be made. Here we shall concentrate on an *abstract* specification (in the manner of abstract interpretation [2]), a *compositional* (or syntax-directed) specification, and a specification using *representations* of expressions. We shall only be concerned with specifying how to *check* that a proposed solution is indeed acceptable; the existence of "best" (or least) acceptable solutions is treated in [9,5,11]. Also we shall only deal with *succinct* specifications as they exhibit a logical flavour that is well suited for semantic considerations.

4.1 Abstract Specification

The most general approach is motivated by the considerations of the collecting semantics (static semantics [2]) in abstract interpretation and works well for open systems. Here the judgement

$$\hat{\rho} \overset{A}{\models} e : W$$

expresses that the pair $(\hat{\rho}, W)$ is an acceptable analysis for the expression e: the W component describes the set of function abstractions that e could result in and the $\hat{\rho}$ component describes the set of function abstractions that the variables inside e could be bound to. It is defined by the following clauses:

$$\hat{\rho} \overset{A}{\models} x : W \quad \textbf{iff} \quad \hat{\rho}(x) \subseteq W$$

$$\hat{\rho} \overset{A}{\models} \texttt{fn}\ x\ \texttt{=>}\ e : W \quad \textbf{iff} \quad \{\texttt{fn}\ x\ \texttt{=>}\ e\} \subseteq W$$

$$\hat{\rho} \overset{A}{\models} e_1\ e_2 : W \quad \textbf{iff} \quad \exists W_1, W_2 : \hat{\rho} \overset{A}{\models} e_1 : W_1 \wedge \hat{\rho} \overset{A}{\models} e_2 : W_2 \wedge$$
$$\forall(\texttt{fn}\ x\ \texttt{=>}\ e') \in W_1 : \exists W' : W_2 \subseteq \hat{\rho}(x) \ \wedge\ \hat{\rho} \overset{A}{\models} e' : W' \ \wedge\ W' \subseteq W$$

The axiom for variables is straightforward: since the abstract environment records the set of abstract values that the variable can be bound to, we just ensure that this set is part of the set of abstract variables that can result from the expression. Also the axiom for function abstractions is straightforward: a function abstraction gives rise to a single abstract value and we ensure that it is part of what can result from the expression. The rule for function applications is a bit more complex. First we must verify that the analysis is correct as regards the operator part and as regards the operand part. For each possible function being applied, we then "link" the abstract values for the actual parameter to those for the formal parameter, we verify that the analysis is correct as regards the body of the function being called, and finally we "link" the abstract values from the function body into those of the application itself.

This definition is appropriate for *open systems* because at the function application we analyse the body of the function actually being called rather than assuming that it is part of the program in question and that it has already

been analysed. Hence the functions called can be allowed to come from the environment, e.g. from a library or from the arguments being supplied to the program in question.

The need for coinduction

There remains the problem of ensuring that the clauses displayed above do in fact define a relation, $\hat{\rho} \models e : W$, for each expression e. This is complicated by the fact that the definition is not syntax-directed: in the clause for function application we perform an analysis of an expression that need not be a subexpression of the function application in question.

The remedy is standard: we need to interpret the clauses coinductively [9]. To do so we regard the clauses as defining a function

$$\mathcal{S} : \mathcal{P}(\mathbf{AEnv} \times \mathbf{Exp} \times \mathbf{AVal}) \to \mathcal{P}(\mathbf{AEnv} \times \mathbf{Exp} \times \mathbf{AVal})$$

that operates over sets of triples of the form $(\hat{\rho}, e, W)$ where $\hat{\rho} \in \mathbf{AEnv}$, $e \in \mathbf{Exp}$, $W \in \mathbf{AVal}$, $\mathbf{AEnv} = \mathbf{Var} \to \mathbf{AVal}$, $\mathbf{AVal} = \mathcal{P}(\mathbf{Exp}^{fn})$ and $\mathbf{Exp}^{fn}$ is the set of function abstractions in $\mathbf{Exp}$. The result of $\mathcal{S}(S)$ is defined by combining the effect of all the clauses except that any occurrence of $\hat{\rho}' \models e' : W'$ on the right hand side is replaced by $(\hat{\rho}', e', W') \in S$. We shall omit the detailed definition of $\mathcal{S}$ but merely note that the function $\mathcal{S}$ is monotonic. By Tarski's Theorem [17] it follows that $\mathcal{S}$ has a complete lattice of fixed points. The least fixed point corresponds to the standard inductive interpretation of the clauses whereas the greatest fixed point corresponds to a coinductive definition [8,3].

By taking the coinductive interpretation of the clauses above we obtain the desired definition of $\hat{\rho} \models e : W$. By reasoning similar to the one in [9] it follows that there always exists a least (or best) analysis that is acceptable in the manner of the above clauses; in particular this means that all programs can be analysed as should not be surprising since one can simply pretend that all function abstractions can reach all places.

The extended language

Let us now pause a moment and think ahead. In case the analysis is to be validated with respect to the environment based semantics we will likely have to analyse also the extensions to the syntax. This calls for adding the following clauses

$$\hat{\rho} \overset{A}{\models} \texttt{close (fn } x \texttt{ => } e\texttt{) in } \rho : W \quad \textbf{iff} \quad \{\texttt{fn } x \texttt{ => } e\} \subseteq W \ \wedge \ \rho \, \mathcal{R}^A \, \hat{\rho}$$

$$\hat{\rho} \overset{A}{\models} \texttt{bind } \rho \texttt{ in } ie' : W \quad \textbf{iff} \quad \exists W' : \hat{\rho} \overset{A}{\models} ie' : W' \ \wedge \ W' \subseteq W \ \wedge \ \rho \, \mathcal{R}^A \, \hat{\rho}$$

(as well as changing the $e_1 \ e_2$ above to be $ie_1 \ ie_2$). The auxiliary relation $\mathcal{R}^A$ ensures that the entities encapsulated in the environments have been properly analysed and it is defined by:

$$\rho \, \mathcal{R}^A \, \hat{\rho} \quad \textbf{iff} \quad \forall x \in \mathsf{dom}(\rho) : \forall y_x, e_x, \rho_x :$$

$$(\rho(x) = \texttt{close (fn } y_x \texttt{ => } e_x\texttt{) in } \rho_x) \Rightarrow$$
$$(\{\texttt{fn } y_x \texttt{ => } e_x\} \subseteq \hat{\rho}(x) \ \wedge \ \rho_x \ \mathcal{R}^A \ \hat{\rho})$$

It is immediate that the auxiliary relation $\mathcal{R}^A$ is well-defined: in each recursive call the environment gets smaller. The overall collection of clauses is then interpreted coinductively as before.

Semantic correctness

Let us now consider the possibility of proving the specification of the analysis correct. Recall that this takes the form of a subject reduction result:

if the analysis $\hat{\rho}$ is acceptable for the expression e_1
and if e_1 evolves into e_2
then the analysis $\hat{\rho}$ is acceptable for the expression e_2.

(Clearly the detailed formulations depend on the semantics used; they will be spelled out in detail as part of the proofs.)

Proposition 4.1 *The possibility of proving the analysis correct with respect to the semantics is given by the following table:*

	$\overset{A}{\models}$
$\xrightarrow{S}$	*no*
$\xrightarrow{E}$	*yes*
$\xrightarrow{\alpha}$	*no*

Proof. In each case we must begin with clearly formulating the correctness statement and then either disprove it by means of an example or conduct a formal proof. For the substitution based semantics the subject reduction result reads as follows:

$$\forall \hat{\rho}, W, e_1, e_2 : (\hat{\rho} \overset{A}{\models} e_1 : W \ \wedge \ e_1 \xrightarrow{S} e_2) \Rightarrow (\hat{\rho} \overset{A}{\models} e_2 : W)$$

To disprove this we shall take $e_1 =$ (`fn x => (fn y => y) (fn z => x)`) (`fn u => u`), $e_2 =$ (`fn y => y`) (`fn z => (fn u => u)`), $\hat{\rho}$(`x`) = {`fn u => u`}, $\hat{\rho}$(`y`) = {`fn z => x`}, $\hat{\rho}$(`z`) = $\emptyset$, $\hat{\rho}$(`u`) = $\emptyset$, and W = {`fn z => x`}.

For the environment based semantics the subject reduction result reads as follows:

$$\forall \hat{\rho}, W, ie_1, ie_2, \rho : (\hat{\rho} \overset{A}{\models} ie_1 : W \ \wedge \ \rho \ \mathcal{R}^A \ \hat{\rho} \ \wedge \ \rho \vdash ie_1 \xrightarrow{E} ie_2) \Rightarrow (\hat{\rho} \overset{A}{\models} ie_2 : W)$$

We proceed by induction on $\rho \vdash ie_1 \xrightarrow{E} ie_2$. In several cases we make use of the lemma

if $\hat{\rho} \overset{A}{\models} e : W_1$ and $W_1 \subseteq W_2$ then $\hat{\rho} \overset{A}{\models} e : W_2$

that can be proved by inspecting each of the clauses defining $\overset{A}{\models}$ in turn.

The negative result for $\xrightarrow{\alpha}$ holds for $\xrightarrow{S\alpha}$ as well as $\xrightarrow{E\alpha}$ (assuming that the correctness statements are analogues of those displayed above). In the case of $\xrightarrow{S\alpha}$ one takes $e_1 = (\texttt{fn x => (fn y => y) (fn z => z)})\ (\texttt{fn u => u})$, $e_2 = (\texttt{fn y => y})\ (\texttt{fn v => v})$, $\hat{\rho}(\texttt{x}) = \{\texttt{fn u => u}\}$, $\hat{\rho}(\texttt{y}) = \{\texttt{fn z => z}\}$, $\hat{\rho}(\texttt{z}) = \emptyset$, $\hat{\rho}(\texttt{u}) = \emptyset$, $\hat{\rho}(\texttt{v}) = \emptyset$, and $W = \{\texttt{fn z => z}\}$. The proof in the case of $\xrightarrow{E\alpha}$ is similar. □

Remark
Given the lemma in the proof of Proposition 4.1 it might seem that one could simplify the clauses for $\models$ by not writing all constraints explicitly in all clauses: one could consider adding a clause saying that $\hat{\rho} \models e : W_2$ **iff** $\exists W_1 : W_1 \subseteq W_2 \wedge \hat{\rho} \models e : W_1$ and then the clause for function abstractions would simply read $\hat{\rho} \models \texttt{fn}\ x\ \texttt{=>}\ e : \{\texttt{fn}\ x\ \texttt{=>}\ e\}$ and "similarly" for the other clauses. This is indeed a common trick used in (inductively defined) type systems but unfortunately it turns out to be problematic for the (coinductively defined) clauses here. The reason is that the coinductive interpretation of the revised definition of $\models$ yields the relation that is universally true (because of the ability to take $W_1 = W_2$)! – So to allow the simplifications (as is done in [11]) one needs a more sophisticated way of interpreting the clauses (and the offending clause must not be added).

4.2 Compositional Specification

In order to obtain a specification that is readily implementable one usually needs to proceed in a more syntax-directed manner. This amounts to checking the bodies of functions when they are "defined" rather than when they are called. One problem with this approach is that we may then end up analysing the bodies of functions that are never called; this can be remedied by adding a reachability component to the analysis (see e.g. [5]) but for conciseness of the presentation we shall abstain from doing so. Another problem with this approach is that we then confine the attention to *closed systems*: we cannot deal with functions that are not part of the program in question (or some *a priori* given library or set of arguments to the program).

Analysing function bodies when "defined" then necessitates an additional component to the analysis: a mechanism for ensuring that the set of function abstractions that can result from the body will be known at all relevant application points. Since we have assumed that all function abstractions have initially been alpha-renamed to have distinct formal parameters, it makes sense to use the formal parameter as the unique identifier for the function abstraction in question. We then extend the global analysis information, $\hat{\rho}$, to contain a component $\hat{\rho}(x\texttt{=>}) = W$ whenever the body of $\texttt{fn}\ x\ \texttt{=>}\ e$ may yield W (just as $\hat{\rho}(x) = W$ whenever the formal parameter of $\texttt{fn}\ x\ \texttt{=>}\ e$ may be bound to abstract values from W).

With these preparations we can then define a judgement of the form

$\hat{\rho} \overset{C}{\models} e : W$

for expressing that the pair $(\hat{\rho}, W)$ is an acceptable analysis for the expression e and bearing in mind that the domain of $\hat{\rho}$ is larger than in the abstract specification. The judgement is defined by the following clauses:

$$\hat{\rho} \overset{C}{\models} x : W \ \textbf{iff} \ \hat{\rho}(x) \subseteq W$$

$$\hat{\rho} \overset{C}{\models} \texttt{fn}\ x\ \texttt{=>}\ e : W \ \textbf{iff} \ \{\texttt{fn}\ x\ \texttt{=>}\ e\} \subseteq W \ \wedge \ \hat{\rho} \overset{C}{\models} e : \hat{\rho}(x\texttt{=>})$$

$$\hat{\rho} \overset{C}{\models} e_1\ e_2 : W \ \textbf{iff} \ \exists W_1, W_2 : \hat{\rho} \overset{C}{\models} e_1 : W_1 \wedge \hat{\rho} \overset{C}{\models} e_2 : W_2 \wedge \forall(\texttt{fn}\ x\ \texttt{=>}\ e') \in W_1 : W_2 \subseteq \hat{\rho}(x) \ \wedge \ \hat{\rho}(x\texttt{=>}) \subseteq W$$

Unlike the abstract specification there is no need to rely on a coinductive definition because the specification is purely compositional (or syntax-directed); however, there is no harm in viewing the specification as being a coinductive definition because the coinductive and inductive definitions turn out to agree (and on philosophical grounds one might indeed argue that one should continue to stress the fact that the specification is coinductive)!

The extended language

Looking ahead to possibly using the environment based semantics for validating the analysis there once more is the need to analyse the extensions to the syntax. This calls for adding the following clauses:

$$\hat{\rho} \overset{C}{\models} \texttt{close}\ (\texttt{fn}\ x\ \texttt{=>}\ e)\ \texttt{in}\ \rho : W \ \textbf{iff} \ \{\texttt{fn}\ x\ \texttt{=>}\ e\} \subseteq W \wedge \rho\, \mathcal{R}^C\, \hat{\rho} \wedge \hat{\rho} \overset{C}{\models} e : \hat{\rho}(x\texttt{=>})$$

$$\hat{\rho} \overset{C}{\models} \texttt{bind}\ \rho\ \texttt{in}\ ie' : W \ \textbf{iff} \ \exists W' : \hat{\rho} \overset{C}{\models} ie' : W' \ \wedge \ W' \subseteq W \ \wedge \ \rho\, \mathcal{R}^C\, \hat{\rho}$$

(as well as changing the $e_1\ e_2$ above to be $ie_1\ ie_2$). The auxiliary relation $\mathcal{R}^C$ is defined as follows:

$$\rho\, \mathcal{R}^C\, \hat{\rho} \ \textbf{iff} \ \forall x \in \mathsf{dom}(\rho) : \forall y_x, e_x, \rho_x : (\rho(x) = \texttt{close}\ (\texttt{fn}\ y_x\ \texttt{=>}\ e_x)\ \texttt{in}\ \rho_x) \ \Rightarrow \ (\{\texttt{fn}\ y_x\ \texttt{=>}\ e_x\} \subseteq \hat{\rho}(x) \ \wedge \ \rho_x\, \mathcal{R}^C\, \hat{\rho} \ \wedge \ \hat{\rho} \overset{C}{\models} e_x : \hat{\rho}(y_x\texttt{=>}))$$

It is now slightly more tricky to ensure that the analysis and the auxiliary relation are well-defined since they depend recursively upon one another. One possibility is to use mathematical induction on n to prove that $\hat{\rho} \models e : W$ and $\rho\, \mathcal{R}^C\, \hat{\rho}$ are well-defined whenever the size of e and ρ is at most n; here the size may be taken to be the finite number of ASCII characters needed to represent the entity.

Relationship between the specifications

We said above that the abstract and compositional specifications differ because we did not include a reachability component. Indeed, once this is done along the lines of [5] one can establish an equivalence result between the two

specifications. To give the flavour of this result we state without proof the following weaker fact that holds for the analyses as defined here; it says that (for closed systems) all acceptable analyses with respect to the compositional specification are also acceptable with respect to the abstract specification.

Fact 4.2 *Let e_* be the given program, let $\boldsymbol{Exp}_*$ be the set of subexpressions of e_* and let $\boldsymbol{Exp}_*^{fn}$ be the set of function abstractions in e_*. Assuming that all $\hat{\rho}(x)$ and W are restricted to be subsets of $\boldsymbol{Exp}_*^{fn}$, i.e. $\boldsymbol{AVal} = \mathcal{P}(\boldsymbol{Exp}_*^{fn})$, we have*

$$\hat{\rho} \overset{A}{\models} e_* : W \;\Leftarrow\; \hat{\rho} \overset{C}{\models} e_* : W.$$

The opposite implication need not hold; as an example take $\hat{\rho}(\mathtt{x}) = \emptyset$, $\hat{\rho}(\mathtt{y}) = \emptyset$, $\hat{\rho}(\mathtt{z}) = \emptyset$, $W = \emptyset$ and consider $e_* = (\mathtt{fn\ x\ =>\ (fn\ y\ =>\ y)\ (fn\ z\ =>\ z)})$.

Semantic correctness

Let us now consider the possibility of establishing a subject reduction result for this analysis.

Proposition 4.3 *The possibility of proving the analysis correct with respect to the semantics is given by the following table:*

	$\overset{C}{\models}$
$\xrightarrow{S}$	*no*
$\xrightarrow{E}$	*yes*
$\xrightarrow{\alpha}$	*no*

Proof. For the substitution based semantics the subject reduction result reads as follows:

$$\forall \hat{\rho}, W, e_1, e_2 : (\hat{\rho} \overset{C}{\models} e_1 : W \;\wedge\; e_1 \xrightarrow{S} e_2) \Rightarrow (\hat{\rho} \overset{C}{\models} e_2 : W)$$

To disprove this statement we proceed as in the proof of Proposition 4.1.

For the environment based semantics the subject reduction result reads as follows:

$$\forall \hat{\rho}, W, ie_1, ie_2, \rho : (\hat{\rho} \overset{C}{\models} ie_1 : W \;\wedge\; \rho\, \mathcal{R}^C\, \hat{\rho} \;\wedge\; \rho \vdash ie_1 \xrightarrow{E} ie_2) \Rightarrow (\hat{\rho} \overset{C}{\models} ie_2 : W)$$

We proceed by induction on $\rho \vdash ie_1 \xrightarrow{E} ie_2$. In several cases we make use of the lemma

$$\text{if } \hat{\rho} \overset{C}{\models} e : W_1 \text{ and } W_1 \subseteq W_2 \text{ then } \hat{\rho} \overset{C}{\models} e : W_2$$

that can be proved by inspecting each of the clauses defining $\overset{C}{\models}$ in turn.

The negative result for $\xrightarrow{\alpha}$ holds for $\xrightarrow{S\alpha}$ as well as $\xrightarrow{E\alpha}$ (assuming that the correctness statements are analogues of those displayed above); it may be proved as in the proof of Proposition 4.1. □

4.3 Representations of Expressions

In terms of implementing the compositional analysis it would seem that we are carrying a lot of useless baggage around: the complete function bodies. This suggests defining a modified analysis that just uses representations of functions. Given our assumption that all function abstractions in the given program have initially been alpha-renamed so as to have distinct formal parameters, it makes sense to let $\texttt{fn}\ x$ serve as a representation of $\texttt{fn}\ x\ \texttt{=>}\ e$. We then define the judgement

$$\hat{\rho} \overset{CR}{\models} e : W$$

by the following clauses that are rather directly obtained from those for $\overset{C}{\models}$:

$$\hat{\rho} \overset{CR}{\models} x : W \ \textbf{iff}\ \hat{\rho}(x) \subseteq W$$

$$\hat{\rho} \overset{CR}{\models} \texttt{fn}\ x\ \texttt{=>}\ e : W \ \textbf{iff}\ \{\texttt{fn}\ x\} \subseteq W \ \wedge\ \hat{\rho} \overset{CR}{\models} e : \hat{\rho}(x\texttt{=>})$$

$$\hat{\rho} \overset{CR}{\models} e_1\ e_2 : W \ \textbf{iff}\ \exists W_1, W_2 : \hat{\rho} \overset{CR}{\models} e_1 : W_1 \wedge \hat{\rho} \overset{CR}{\models} e_2 : W_2 \wedge$$
$$\forall(\texttt{fn}\ x) \in W_1 : W_2 \subseteq \hat{\rho}(x) \ \wedge\ \hat{\rho}(x\texttt{=>}) \subseteq W$$

The extended language

The clauses relevant for the extensions to the syntax are minor variations of those considered before:

$$\hat{\rho} \overset{CR}{\models} \texttt{close}\ (\texttt{fn}\ x\ \texttt{=>}\ e)\ \texttt{in}\ \rho : W \ \textbf{iff}\ \{\texttt{fn}\ x\} \subseteq W \ \wedge\ \rho\ \mathcal{R}^{CR}\ \hat{\rho} \ \wedge\ \hat{\rho} \overset{CR}{\models} e : \hat{\rho}(x\texttt{=>})$$

$$\hat{\rho} \overset{CR}{\models} \texttt{bind}\ \rho\ \texttt{in}\ ie' : W \ \textbf{iff}\ \exists W' : \hat{\rho} \overset{CR}{\models} ie' : W' \ \wedge\ W' \subseteq W \ \wedge\ \rho\ \mathcal{R}^{CR}\ \hat{\rho}$$

Finally, the definition of the auxiliary relation $\mathcal{R}^{CR}$ is obtained from the one for $\mathcal{R}^{C}$:

$$\rho\ \mathcal{R}^{CR}\ \hat{\rho} \ \textbf{iff}\ \forall x \in \mathsf{dom}(\rho) : \forall y_x, e_x, \rho_x :$$
$$(\rho(x) = \texttt{close}\ (\texttt{fn}\ y_x\ \texttt{=>}\ e_x)\ \texttt{in}\ \rho_x) \ \Rightarrow$$
$$(\{\texttt{fn}\ y_x\} \subseteq \hat{\rho}(x) \ \wedge\ \rho_x\ \mathcal{R}^{CR}\ \hat{\rho} \ \wedge\ \hat{\rho} \overset{CR}{\models} e_x : \hat{\rho}(y_x\texttt{=>}))$$

Well-definedness of these definitions follows as for the compositional specification.

Relationship between the specifications

Intuitively the two compositional specifications should be equivalent because the body of a function abstraction is not used to carry any information. We state without proof the following fact; it states an equivalence result for closed systems.

Fact 4.4 *Let e_* be the given program, let $\boldsymbol{Exp}_*$ be the set of subexpressions of e_* and let $\boldsymbol{Exp}_*^{fn}$ be the set of function abstractions in e_*. Writing $\mathsf{ret}(W) = \{(\texttt{fn}\ x) \mid (\texttt{fn}\ x\ \texttt{=>}\ e) \in W\}$ and assuming that all $\hat{\rho}(x)$ and W are restricted*

to be subsets of $\boldsymbol{Exp}_*^{fn}$, *i.e.* $\boldsymbol{AVal} = \mathcal{P}(\boldsymbol{Exp}_*^{fn})$, *we have*

$$\hat{\rho} \overset{C}{\models} e_* : W \;\Rightarrow\; (ret \circ \hat{\rho}) \overset{CR}{\models} e_* : ret(W)$$

Furthermore, writing $exp(W) = \{(\texttt{fn}\ x\ \texttt{=>}\ e) \in \boldsymbol{Exp}_* \mid (\texttt{fn}\ x) \in W\}$ *and assuming that all* $\hat{\rho}(x)$ *and* W *are restricted to be subsets of* $ret(\boldsymbol{Exp}_*^{fn})$, *i.e.* $\boldsymbol{AVal} = \mathcal{P}(ret(\boldsymbol{Exp}_*^{fn}))$, *we have*

$$(exp \circ \hat{\rho}) \overset{C}{\models} e_* : exp(W) \;\Leftarrow\; \hat{\rho} \overset{CR}{\models} e_* : W$$

Note that $exp(W)$ produces exactly the same number of elements as in W given our assumption that all function abstractions have initially been alpha-renamed to have distinct formal parameters.

Semantic correctness

The use of representations of expressions rather the expressions themselves turns out to facilitate establishing an analogoue of a subject reduction result that defeated us earlier.

Proposition 4.5 *The possibility of proving the analysis correct with respect to the semantics is given by the following table:*

	$\overset{CR}{\models}$
$\xrightarrow{S}$	*yes*
$\xrightarrow{E}$	*yes*
$\xrightarrow{\alpha}$	*no*

Proof. For the substitution based semantics the subject reduction reads as follows:

$$\forall \hat{\rho}, W, e_1, e_2 : (\hat{\rho} \overset{CR}{\models} e_1 : W \;\wedge\; e_1 \xrightarrow{S} e_2) \Rightarrow (\hat{\rho} \overset{CR}{\models} e_2 : W)$$

The proof is by induction on $e_1 \xrightarrow{S} e_2$. For $(\texttt{fn}\ x\ \texttt{=>}\ e)\ v_2 \xrightarrow{S} e[x \mapsto v_2]$ we use the lemma

$$\text{if } \hat{\rho} \overset{CR}{\models} e : W_1 \text{ and } W_1 \subseteq W_2 \text{ then } \hat{\rho} \overset{CR}{\models} e : W_2$$

(that can be proved by inspecting each of the clauses defining $\overset{CR}{\models}$ in turn) and also the lemma

$$\text{if } \hat{\rho} \overset{CR}{\models} e : W \text{ and } \hat{\rho} \overset{CR}{\models} v : \hat{\rho}(x) \text{ then } \hat{\rho} \overset{CR}{\models} e[x \mapsto v] : W$$

(that can be proved by structural induction on e) and we also use the fact that no variable capture can take place in $e[x \mapsto v]$ (because if v is a value then $\mathsf{FV}(v) \subseteq \mathsf{FV}(e_*)$ and the set of formal parameters is disjoint from the set of global variables, $\mathsf{FV}(e_*)$).

For the environment based semantics the subject reduction result reads as follows:

$$\forall \hat{\rho}, W, ie_1, ie_2, \rho : (\hat{\rho} \overset{CR}{\models} ie_1 : W \ \wedge \ \rho \, \mathcal{R}^{CR} \, \hat{\rho} \ \wedge \ \rho \vdash ie_1 \xrightarrow{E} ie_2) \Rightarrow (\hat{\rho} \overset{CR}{\models} ie_2 : W)$$

For the proof we proceed as in the proof of the corresponding case in Proposition 4.3.

The negative result for $\xrightarrow{\alpha}$ holds for $\xrightarrow{S\alpha}$ as well as $\xrightarrow{E\alpha}$ (assuming that the correctness statements are analogues of those displayed above); it may be proved as in the proof of Proposition 4.3. □

Representations of expressions for abstract specification
The reader might wonder whether it is only in the case of compositional specifications that it is possible to work with representations of expressions rather than the expressions themselves. In the Appendix we shall show that it is indeed possible to do so for abstract specifications although the resulting specification, $\overset{AR}{\models}$, is not of interest because the analysis is much coarser than the other analyses.

5 Conclusion

Flow Logic is by no means the first approach to formulating program analyses in a logical form. However, in our view it is the first approach that aims at integrating the insights from *existing* program analysis technologies (such as data flow analysis, control flow analysis and abstract interpretation) into a common form that is applicable to a wide variety of programming languages. This then motivates the current investigation into the relative usefulness of different kinds of semantics.

The technical results concerning the possibility of proving the analyses correct may be summarised as follows[2]:

	$\overset{A}{\models}$	$\overset{C}{\models}$	$\overset{CR}{\models}$	$\overset{AR}{\models}$
$\xrightarrow{S}$	no	no	yes	(yes)
$\xrightarrow{E}$	yes	yes	yes	(yes)
$\xrightarrow{\alpha}$	no	no	no	

Here $\overset{C}{\models}$ and $\overset{CR}{\models}$ are equally precise (Fact 4.4) and only "slightly coarser" than $\overset{A}{\models}$ (Fact 4.2) whereas $\overset{AR}{\models}$ is so coarse as to be of no interest (Fact A.1). The table clearly shows that the environment based semantics is more flexible than either substitution based semantics or semantics making use of structural congruences[3] (like alpha-renaming) in terms of being able to accomodate a

[2] We refer to the Appendix for the missing entry.

[3] Current work on the π-calculus studies techniques aimed at overcoming some of these difficulties; this involves changing the syntax of the language so as to contain "markers" for all entities that are not invariant under the congruence.

variety of specification styles for program analysis.

In our view it is easiest to develop a correct and useful program analysis if one proceeds as follows:

- *begin by developing an abstract specification.*

This is particularly so for novel calculi involving distribution and mobility because abstract specifications are able to deal with open systems. Also one is less likely to fail to observe that the compositional specifications restrict themselves to closed systems and that a reachability component is needed in order to obtain the same precision as in the abstract specification.

In order to establish semantic correctness by means of a subject reduction result it is a general principle that:

- *the analysis information should remain invariant under evaluation.*

As we have seen this puts severe demands on the choice of operational semantics: one is more or less forced to abandon working with a simple substitution based semantics in order to work with a more complex environment based semantics; unfortunately, (in keeping with [13]) this requires "artificial" extensions to the syntax, that then also have to be analysed thereby reducing the level of abstraction of the reasoning.

We believe that a compositional (or syntax-directed) specification is a prerequisite for obtaining an efficient implementation. As was explained above, this involves restricting the attention to closed systems. The development in Section 4.2 is semi-compositional [7] in the sense that all expressions considered are subexpressions of the given program; however, semi-compositionality does not suffice for having a free choice between using environment based or substitution based semantics. To achieve this we used representations of expressions in Section 4.3.

In this paper we have only considered *succinct* specifications and have ignored the *verbose* formulations of flow logic that are likely to be needed in order to obtain an efficient implementation. One further principle worth stating is that:

- *explicit program points (in the form of labelling all subexpressions [9] or demanding all expressions to be in "A-normalform" [4]) are needed for verbose formulations but not for succinct specifications.*

For the succinct formulations considered in this paper we merely assumed that all function abstractions had initially been alpha-renamed so as to have distinct formal parameters that were also distinct from the global variables. We refer to [11] for how to transform a succinct specification into a more verbose specification and to [5] for an example of how a verbose specification may be implemented.

Acknowledgements

This research was supported in part by the DART-AROS project funded by the Danish Research Councils. We wish to thank Chiara Bodei, Pierpaolo Degano, Chris Hankin, Carolyn Talcott and Mitch Wand for discussions about semantics based program analysis; Torben Amtoft provided helpful comments on a previous version.

References

[1] H. P. Barendregt. *The Lambda Calculus — Its Syntax and Semantics*, volume 103 of *Studies in logic and the foundation of mathematics*. North-Holland, revised edition, 1984.

[2] P. Cousot and R. Cousot. Abstract Interpretation: a Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Proc. 4th POPL*, pages 238–252. ACM Press, 1977.

[3] P. Cousot and R. Cousot. Inductive definitions, semantics and abstract interpretation. In *Proc. POPL '92*, pages 83–94. ACM Press, 1992.

[4] C. Flanagan and M. Felleisen. The semantics of future and its use in program optimization. In *Proc. POPL '95*, pages 209–220. ACM Press, 1995.

[5] K. L. S. Gasser, F. Nielson, and H. R. Nielson. Systematic realisation of control flow analyses for CML. In *Proceedings of ICFP'97*, pages 38–51. ACM Press, 1997.

[6] N. Heintze. Set-based analysis of ML programs. In *Proc. LFP'94*, pages 306–317, 1994.

[7] N. D. Jones. What not to do when writing an interpreter for specialisation. In *Partial Evaluation*, pages 216–237. Springer Lecture Notes in Computer Science 1110, 1996.

[8] R. Milner. *Communication and Concurrency*. Prentice Hall International, 1989.

[9] F. Nielson and H. R. Nielson. Infinitary Control Flow Analysis: a Collecting Semantics for Closure Analysis. In *Proc. POPL '97*, 1997.

[10] F. Nielson, H. R. Nielson, and C. L. Hankin. *Principles of Program Analysis: Flows and Effects*. To appear in 1999.

[11] H. R. Nielson and F. Nielson. Flow logics for constraint based analysis. In *Proc. CC'98*, to appear in Springer Lecture Notes in Computer Science, 1998.

[12] J. Palsberg. Closure analysis in constraint form. *ACM TOPLAS*, 17 (1):47–62, 1995.

[13] G. D. Plotkin. A structural approach to operational semantics. Technical Report FN-19, DAIMI, Aarhus University, Denmark, 1981.

[14] P. Sestoft. *Analysis and efficient implementation of functional programs.* PhD thesis, Department of Computer Science, University of Copenhagen, Denmark, 1991.

[15] O. Shivers. Control flow analysis in Scheme. In *Proc. PLDI'88*, pages 164–174. ACM Sigplan Notices 7 (1), 1988.

[16] O. Shivers. The semantics of Scheme control-flow analysis. In *Partial Evaluation and Semantics-Based Program Manipulation.* ACM SIGPLAN Notices 26 (9), 1991.

[17] A. Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific J. Math.*, 5:285–309, 1955.

A Representations of Expressions for Abstract Specification

In this appendix we show that it is possible to use representations of expressions also for abstract specifications, although the resulting analysis is so coarse as to be of little interest.

So let us define a judgement

$$\hat{\rho} \overset{AR}{\models} e : W$$

by the following clauses:

$$\hat{\rho} \overset{AR}{\models} x : W \quad \textbf{iff} \quad \hat{\rho}(x) \subseteq W$$

$$\hat{\rho} \overset{AR}{\models} \texttt{fn}\ x\ \texttt{=>}\ e : W \quad \textbf{iff} \quad \{\texttt{fn}\ x\} \subseteq W$$

$$\hat{\rho} \overset{AR}{\models} e_1\ e_2 : W \quad \textbf{iff} \quad \exists W_1, W_2 : \hat{\rho} \overset{AR}{\models} e_1 : W_1 \wedge \hat{\rho} \overset{AR}{\models} e_2 : W_2 \wedge$$
$$\forall(\texttt{fn}\ x) \in W_1 : \forall e' : \exists W' : W_2 \subseteq \hat{\rho}(x) \ \wedge\ \hat{\rho} \overset{AR}{\models} e' : W' \ \wedge\ W' \subseteq W$$

For completeness sake we also list the clauses for the extended syntax:

$$\hat{\rho} \overset{AR}{\models} \texttt{close}\ (\texttt{fn}\ x\ \texttt{=>}\ e)\ \texttt{in}\ \rho : W \quad \textbf{iff} \quad \{\texttt{fn}\ x\} \subseteq W \ \wedge\ \rho\ \mathcal{R}^{AR}\ \hat{\rho}$$

$$\hat{\rho} \overset{AR}{\models} \texttt{bind}\ \rho\ \texttt{in}\ ie' : W \quad \textbf{iff} \quad \exists W' : \hat{\rho} \overset{AR}{\models} ie' : W' \ \wedge\ W' \subseteq W \ \wedge\ \rho\ \mathcal{R}^{AR}\ \hat{\rho}$$

The definition of the auxiliary relation then is:

$$\rho\ \mathcal{R}^{AR}\ \hat{\rho} \quad \textbf{iff} \quad \forall x \in \mathsf{dom}(\rho) : \forall y_x, e_x, \rho_x :$$
$$(\rho(x) = \texttt{close}\ (\texttt{fn}\ y_x\ \texttt{=>}\ e_x)\ \texttt{in}\ \rho_x) \ \Rightarrow\ (\{\texttt{fn}\ y_x\} \subseteq \hat{\rho}(x) \wedge \rho_x\ \mathcal{R}^{AR}\ \hat{\rho})$$

Relationship between the specifications

The following result shows that the only acceptable analysis for unevaluated programs is the one that says that *all* function abstractions can reach *all* places. For the formal statement we need a few preparations. First note that an expression in the original syntax is either an application, a variable or a

function abstraction; if it is an application it can be "maximally expanded" into one of the forms $(((x\ e_1)\ e_2)\ \cdots\ e_n)$ or $((((\texttt{fn}\ y\ \texttt{=>}\ e)\ e_1)\ e_2)\ \cdots\ e_n)$ (for $n > 0$). To get access to the variable x occurring to the very left, if indeed such a variable exists, we shall define the set $\mathsf{LV}(e_*)$ as follows: $\mathsf{LV}(x) = \{x\}$, $\mathsf{LV}(\texttt{fn}\ x\ \texttt{=>}\ e) = \emptyset$ and $\mathsf{LV}(e_1\ e_2) = \mathsf{LV}(e_1)$. Clearly $\mathsf{LV}(e_*)$ contains at most one element.

Fact A.1 *Let e_* be the given program and suppose that it is an application (i.e. it is not a variable or a function abstraction), that $\forall x \in \mathsf{LV}(e_*) : \hat{\rho}(x) \neq \emptyset$ and that (for all x) $\hat{\rho}(x)$ and W are restricted to be subsets of $\mathsf{ret}(\boldsymbol{Exp}^{fn})$ where $\mathsf{ret}(W) = \{(\texttt{fn}\ x) \mid (\texttt{fn}\ x\ \texttt{=>}\ e) \in W\}$ and $\boldsymbol{Exp}^{fn}$ is the set of function abstractions;*

$$\text{if } \hat{\rho} \overset{AR}{\models} e_* : W \text{ then } \forall x : \hat{\rho}(x) = \mathsf{ret}(\boldsymbol{Exp}^{fn}) \text{ and } W = \mathsf{ret}(\boldsymbol{Exp}^{fn}).$$

Proof (sketch). The key to the proof is that in the clause for application we can choose $e' = (\texttt{fn}\ x_i\ \texttt{=>}\ x_i)\ e_j$ where x_i ranges through all variables and e_j ranges through all function abstractions. For the proof note that since e_* is an application it can be "maximally expanded": either into $(((x\ e_1)\ e_2)\ \cdots\ e_n)$ (for $n > 0$) in which case we know that $\hat{\rho}(x) \neq \emptyset$, or else into $((((\texttt{fn}\ y\ \texttt{=>}\ e)\ e_1)\ e_2)\ \cdots\ e_n)$ (for $n > 0$); in both cases we prove the desired result by induction on n. □

Semantic correctness

Despite our lack of interest in this analysis let us nonetheless consider the possibility of establishing a subject reduction result; in the table below we put answers in parantheses in order to remind us that the analysis is substantially coarser than those previously considered.

Proposition A.2 *The possibility of proving the analysis correct with respect to the semantics is given by the following table:*

	$\overset{AR}{\models}$
$\xrightarrow{S}$	*(yes)*
$\xrightarrow{E}$	*(yes)*
$\xrightarrow{S\alpha}$	*(yes)*
$\xrightarrow{E\alpha}$	*(no)*

Proof (sketch). The formulations of the subject reduction results are much as in the proofs of Proposition 4.1; we shall dispense with repeating them here.

We first consider the case of the substitution based semantics. One way to prove the result is to proceed as in the proof of the corresponding case in Proposition 4.5. — A more "abstract" way of proving the result proceeds as follows where we must take care to restrict all $\hat{\rho}(x)$ and W to be subsets

of $\mathsf{ret}(\mathbf{Exp}^{fn})$. If an expression (in the original syntax) can evaluate into another then it must be an application and hence it must expand into either $(((x\ e_1)\ e_2)\ \cdots\ e_n)$ (for $n > 0$) or else $((((\texttt{fn}\ y\ \texttt{=>}\ e)\ e_1)\ e_2)\ \cdots\ e_n)$ (for $n > 0$). In fact the first case cannot arise because variables are not values. This leaves us with the second case where it follows from Fact A.1 that the $\hat{\rho}$ and W in question must state that all function abstractions reach everywhere; but this suffices for analysing an arbitrary expression since all constraints are then vacuously fulfilled.

We next consider the case of the environment based semantics where we proceed as in the proof of the corresponding case in Proposition 4.1.

The positive result for $\xrightarrow{S\alpha}$ may be proved as in the "abstract" way of proving $\xrightarrow{S}$ above: if there is any possibility of using alpha-renaming we know by Fact A.1 that the $\hat{\rho}$ and W in question must state that all function abstractions reach everywhere; but then alpha-renaming is not harmful.

The negative result for $\xrightarrow{E\alpha}$ may be obtained as follows: let $\forall x : \hat{\rho}(x) = \emptyset$, $W = \{\texttt{fn x => x}\}$, $\rho = [\,]$, $ie_1 = \texttt{fn x => x}$ and $ie_2 = \texttt{close (fn y => y) in}\ [\,]$.□

Electronic Notes in Theoretical Computer Science 10 (1998)
URL: http://www.elsevier.nl/locate/entcs/volume10.html 19 pages

An Introduction to History Dependent Automata [1]

Ugo Montanari and Marco Pistore

Computer Science Department
University of Pisa
Corso Italia 40, 56100 Pisa, Italy
{ugo,pistore}@di.unipi.it

Abstract

Automata (or *labeled transition systems*) are widely used as operational models in the field of process description languages like CCS [13]. There are however classes of formalisms that are not modelled adequately by the automata. This is the case, for instance, of the π-calculus [15,14], an extension of CCS where channels can be used as values in the communications and new channels can be created dynamically. Due to the necessity to represent the creation of new channels, infinite automata are obtained in this case also for very simple agents and a non-standard definition of bisimulation is required.

In this paper we present an enhanced version of automata, called *history dependent automata*, that are adequate to represent the operational semantics of π-calculus and of other *history dependent formalisms*. We also define a bisimulation equivalence on history dependent automata, that captures π-calculus bisimulation. The results presented here are discussed in more detail in [21].

1 Introduction

In the context of process algebras (e.g., Milner's CCS [13]), *automata* (or *labeled transition systems*) are often used as operational models. They allow for a simple representation of process behavior and many concepts and theoretical results for these process algebras are independent from the particular syntax of the languages, and can be formulated directly on automata. In particular, this is true for the behavioral equivalences and preorders which have been

[1] Research supported in part by CNR Integrated Project *Metodi e Strumenti per la Progettazione e la Verifica di Sistemi Eterogenei Connessi mediante Reti di Comunicazione* and Esprit Working Group *CONFER2*.

defined for these languages, like bisimulation equivalence [13,22]: in fact they take into account just the labeled actions an agent can perform.

Automata are also important from an algorithmic point of view: efficient and practical techniques and tools for verification [9,12] have been developed for *finite-state* automata. Finite state verification is successful here, differently than in ordinary programming, since the control part and the data part of protocols and hardware components can be often cleanly separated, and the control part is usually both quite complex and finite state.

There are classes of process description languages, however, whose operational semantics is not described in a satisfactory way by ordinary automata. A paradigmatic example is provided by π-calculus [15,14]. This calculus can be considered as a foundational calculus for concurrent functional languages, as λ-calculus for sequential functional languages. In π-calculus channel names can be used as messages in the communications, thus allowing for a dynamic reconfiguration of process acquaintances. More importantly, π-calculus names can model objects (in the sense of object oriented programming [24]) and name sending thus models higher order communication [23]. New channels between the process and the environment can be created at run-time and referred to in subsequent communications.

The operational semantics of π-calculus is given via a labeled transition system. This is not completely adequate to deal with the peculiar features of the calculus and complications arise in the representation of the creation of new channels. Consider process $p = (\nu y)\,\bar{x}y.q$; it communicates name y on channel x and then behaves like q. Channel y is initially a local, restricted channel for process p, however the restriction is removed when the communication takes place, since it makes name y known also outside the process. This communication represents the creation of a new channel. In the ordinary semantics of the π-calculus it is modelled by means of an infinite bunch of transitions of the form $p \xrightarrow{\bar{x}(w)} q\{w/y\}$, where w is any name that is not already in use in p. This way to represent the creation of new names has some disadvantages: first of all, also very simple π-calculus agents, like p, give rise to infinite-state and infinite-branching transition systems. Moreover, equivalent processes do not necessarily have the same sets of channel names; so, there are processes q equivalent to p which cannot use y as the name for the newly created channel. Special rules are hence needed in the definition of bisimulation, which is not the standard one for transition systems, and, as a consequence, standard theories and algorithms do not apply to π-calculus.

This is a general problem for the class of *history-dependent calculi.* A calculus is history dependent if the observations labeling the transitions of an agent may refer to informations — names in the case of π-calculus — generated in previous transitions of the agent.

In [21] *history-dependent automata* (*HD-automata* in brief) are proposed as a general model for history-dependent calculi. As ordinary automata, they are composed of states and of transitions between states. To deal with the

peculiar problems of history-dependent calculi, however, states and transition are enriched with sets of local names: in particular, each transition can refer to the names associated to its source state but can also generate new names, which can then appear in the destination state. In this manner, the names are not global and static, as in ordinary labeled transition systems, but they are explicitly represented within states and transitions and can be dynamically created.

This permits to represent adequately the behavior of history-dependent processes. In particular, π-calculus agents can be translated into HD-automata and a first sign of the adequacy of HD-automata for dealing with π-calculus is that a large class of *finitary* π-calculus agents can be represented by finite-state HD-automata.

In [21] a general definition of bisimulation for HD-automata is also given. An important result is that this general bisimulation equates the HD-automata obtained from two π-calculus agents if and only if the agents are bisimilar according to the ordinary (strong, early) π-calculus bisimilarity relation.

These results do not hold only for the π-calculus: similar mappings exist also for other history-dependent calculi. In previous papers [20,18,19] we defined mappings to HD-automata for CCS with localities [4], for CCS with causality [7,6,11], and, to consider an example outside the field of process algebras, for the history-preserving semantics of Petri nets [2].

Papers [20,18,19] introduce the applications of HD-automata without resorting to categories. Report [21] defines HD-automata and HD-bisimulation both in a set theoretical style and following the uniform categorical approach of [10] based on spans of open maps.

In this paper we summarize some of the results of [21]. In particular, we define HD-automata in a categorical framework, by exploiting a classical categorical definition of ordinary automata. We also show that π-calculus agents can be translated into HD-automata. Finally, we introduce HD-bisimulation by applying to HD-automata the approach of open maps. We refer to [21] for the proof of the results presented here and for a deeper study of the properties of HD-automata.

2 The π-calculus

The π-calculus [15,14] is an extension of CCS in which channel names can be used as values in the communications, i.e., channels are first-order values. This possibility of communicating names gives to the π-calculus a richer expressive power that CCS: in fact it allows to generate dynamically new channels and to change the interconnection structure of the processes. The π-calculus has been successfully used to model object oriented languages [24], and also higher-order communications can be easily encoded in the π-calculus [23], thus allowing for code migration.

Many versions of π-calculus have appeared in the literature. The π-calculus

we present here is *early* and *monadic*; it was first introduced in [16], but we present a slightly simplified version, following in part the style proposed in [23,14] for the polyadic π-calculus.

Let $\mathfrak{N}$ be an infinite, denumerable set of *names*, ranged over by $a, \ldots, z$, and let *Var* be a finite set of *agent identifiers*, denoted by $A, B, \ldots$; the π-calculus *agents*, ranged over by $p, q, \ldots$, are defined by the syntax

$$p ::= \mathbf{0} \mid \pi.p \mid p|p \mid p+p \mid (\nu x)\, p \mid [x{=}y]p \mid A(x_1, \ldots, x_n)$$

where the *prefixes* π are defined by the syntax

$$\pi ::= \tau \mid \bar{x}y \mid x(y).$$

The occurrences of y in $x(y).p$ and $(\nu y)\, p$ are bound; *free names* of agent p are defined as usual and we denote them with $\mathtt{fn}(p)$. For each identifier A there is a definition $A(y_1, \ldots, y_n) \stackrel{\text{def}}{=} p_A$ (with y_i all distinct and $\mathtt{fn}(p_A) \subseteq \{y_1, \ldots, y_n\}$); we assume that, whenever A is used, its arity n is respected. Finally we require that each agent identifier in p_A is in the scope of a prefix (guarded recursion).

If $\sigma : \mathfrak{N} \to \mathfrak{N}$, we denote with $p\sigma$ the agent p whose free names have been replaced according to substitution σ (possibly with changes in the bound names); we denote with $\{y_1/x_1 \cdots y_n/x_n\}$ the substitution that maps x_i into y_i for $i = 1, \ldots, n$ and which is the identity on the other names.

Notice that, with some abuse of notation, we can see substitution σ in $p\sigma$ as a function on $\mathtt{fn}(p)$ rather than on $\mathfrak{N}$; in fact, $p\sigma$ and $p\sigma'$ coincide whenever σ and σ' coincide on $\mathtt{fn}(p)$. So, we say that substitution σ is injective for p if $\sigma : \mathtt{fn}(p) \to \mathfrak{N}$ is an injective function. We also say that agents p and q differ for a bijective substitution if there exists some bijective function $\sigma : \mathtt{fn}(p) \to \mathtt{fn}(q)$ such that $q = p\sigma$.

We define π-calculus agents up to a *structural congruence* $\equiv$, in the style of the Chemical Abstract Machine [1]. This structural congruence allows to identify all the agents which represent essentially the same system and which differ just for syntactical details; moreover it simplifies the presentation of the operational semantics. The structural congruence $\equiv$ is the smallest congruence that respects the following rules.

(alpha) $(\nu x)\, p \equiv (\nu y)\, (p\{y/x\})$ if $y \notin \mathtt{fn}(p)$

(sum) $p+\mathbf{0} \equiv p \quad p+q \equiv q+p \quad p+(q+r) \equiv (p+q)+r$

(par) $p|\mathbf{0} \equiv p \quad p|q \equiv q|p \quad p|(q|r) \equiv (p|q)|r$

(res) $(\nu x)\, \mathbf{0} \equiv \mathbf{0} \quad (\nu x)\, (\nu y)\, p \equiv (\nu y)\, (\nu x)\, p$

$(\nu x)\, (p|q) \equiv p|(\nu x)\, q$ if $x \notin \mathtt{fn}(p)$

(match) $[x = x]p \equiv p \quad [x = y]\mathbf{0} \equiv \mathbf{0}$

By exploiting the structural congruence $\equiv$, each π-calculus agent can be seen as a set of *sequential processes* that act in parallel, sharing a set of channels, some of which are global (unrestricted) whereas some other are local

$$\tau.p \xrightarrow{\tau} p \qquad \bar{x}y.p \xrightarrow{\bar{x}y} p \qquad x(y).p \xrightarrow{xz} p\{z/y\}$$

$$\frac{p_1 \xrightarrow{\alpha} p'}{p_1+p_2 \xrightarrow{\alpha} p'} \qquad \frac{p_1 \xrightarrow{\alpha} p_1'}{p_1|p_2 \xrightarrow{\alpha} p_1'|p_2} \text{ if } \mathtt{bn}(\alpha) \cap \mathtt{fn}(p_2) = \emptyset$$

$$\frac{p_1 \xrightarrow{\bar{x}y} p_1' \quad p_2 \xrightarrow{xy} p_2'}{p_1|p_2 \xrightarrow{\tau} p_1'|p_2'} \qquad \frac{p_1 \xrightarrow{\bar{x}(y)} p_1' \quad p_2 \xrightarrow{xy} p_2'}{p_1|p_2 \xrightarrow{\tau} (\nu y)\,(p_1'|p_2')} \text{ if } y \notin \mathtt{fn}(p_2)$$

$$\frac{p \xrightarrow{\alpha} p'}{(\nu x)\,p \xrightarrow{\alpha} (\nu x)\,p'} \text{ if } x \notin \mathtt{n}(\alpha) \qquad \frac{p \xrightarrow{\bar{x}y} p'}{(\nu y)\,p \xrightarrow{\bar{x}(z)} p'\{z/y\}} \text{ if } x \neq y, z \notin \mathtt{fn}((\nu y)\,p)$$

$$\frac{p_A\{y_1/x_1 \cdots y_n/x_n\} \xrightarrow{\alpha} p'}{A(y_1,\ldots,y_n) \xrightarrow{\alpha} p'} \text{ if } A(x_1,\ldots,x_n) \stackrel{\text{def}}{=} p_A$$

Table 1
Early operational semantics.

(restricted). Each sequential process is a term of the form

$$s ::= \pi.p \;\Big|\; p{+}p \;\Big|\; A(x_1,\ldots,x_n)$$

that can be considered as a "program" describing all the possible behaviors of the sequential process. These sequential processes are then connected by means of the operators of parallel composition and restriction, that allow to describe the structure of the system in which the processes act.

The *actions* an agent can perform are defined by the syntax

$$\alpha ::= \tau \;\Big|\; xy \;\Big|\; \bar{x}y \;\Big|\; \bar{x}(z)$$

and are called respectively *synchronization*, *input*, *free output* and *bound output* actions; x and y are free names of α ($\mathtt{fn}(\alpha)$), whereas z is a bound name ($\mathtt{bn}(\alpha)$); moreover $\mathtt{n}(\alpha) = \mathtt{fn}(\alpha) \cup \mathtt{bn}(\alpha)$. Name x is called the *subject* and y or z the *object* of the action.

The transitions for the *early operational semantics* are defined by the axiom schemata and the inference rules of Table 1.

Some comments on the syntax and on the operational semantics of π-calculus are now in order. The syntax of π-calculus is similar to that of CCS: the most important difference is in the prefixes. The *output* prefix $\bar{x}y.p$ specifies not just the channel x for the communication, but also the value y that is sent on x; in the input prefixes $x(y).p$, name x represents still the channel, whereas y represents a formal variable in p, that is instantiated by the effectively received value when the input transition takes place.

The matching $[x{=}y]p$ represents a guard for agent p: agent p can act only if x and y coincide; this behavior is obtained by exploiting the structural congruence: in fact $[x{=}x]p \equiv p$; no transition can be derived from $[x{=}y]p$ if $x \neq y$.

Notice that, in the case of the π-calculus, the actions a process can perform are different from the prefixes. This happens due to the input and to the bound output. In the case of the input, the prefix has the form $x(y)$, while the

action has the form xz; in fact, y represent a formal variable, whereas z is the effectively received value[2]. The bound output transitions are specific of the π-calculus; they represent the communication of a name that was previously restricted, i.e., it corresponds to the generation of a new channel between the agent and the environment.

Now we present the definition of the early bisimulation for the π-calculus.

Definition 2.1 (early bisimulation) A relation $\mathcal{R}$ over agents is an *early simulation* if whenever $p \mathcal{R} q$ then:

for each $p \xrightarrow{\alpha} p'$ with $\mathtt{bn}(\alpha) \cap \mathtt{fn}(p,q) = \emptyset$ there is some $q \xrightarrow{\alpha} q'$ such that $p' \mathcal{R} q'$.

A relation $\mathcal{R}$ is an *early bisimulation* if both $\mathcal{R}$ and $\mathcal{R}^{-1}$ are early simulations. Two agents p and q are *early bisimilar*, written $p \sim_\pi q$, if $p \mathcal{R} q$ for some early bisimulation $\mathcal{R}$.

As for CCS-like calculi, a labeled transition system is used to give an operational semantics to the π-calculus. However, this way to present the operational semantics has some disadvantages. For instance, an infinite number of transitions correspond even to very simple agents, like $p = x(y).\bar{y}z.\mathbf{0}$: in fact, this agent can perform an infinite number of different input transitions $p \xrightarrow{xw} \bar{w}z.\mathbf{0}$, corresponding to all the possible choices of $w \in \mathfrak{N}$. It is clear that, except for x and z, which are the free names of p, all the other names are indistinguishable as input values for the future behavior of p. However, this fact is not reflected in the operational semantics.

Also consider process $q = (\nu y)\,\bar{x}y.y(z).\mathbf{0}$. It is able to generate a new channel by communicating name y in a bound output. The creation of a new name is represented in the transition system by means of an infinite bunch of transitions $q \xrightarrow{\bar{x}(w)} w(z).\mathbf{0}$, where, in this case, w is any name different from x: the creation of a new channel is modelled by using all the names which are not already in use to represent it. As a consequence, the definition of bisimulation is not the ordinary one: in general two bisimilar process can have different sets free names, and the clause "$\mathtt{bn}(\alpha) \cap \mathtt{fn}(p,q) = \emptyset$" has to be added in Definition 2.1 to deal with those bound output transitions which use a name that is used only in one of the two processes. The presence of this clause makes it difficult to reuse standard theory and algorithms for bisimulation on the π-calculus — see for instance [5].

3 History-dependent automata

As explained in the Introduction, ordinary automata are insufficient to deal with history-dependent calculi. To address this problem, in this section we

[2] This is not true in all the versions of the π-calculus; in the case of the *late* and *open* versions, for instance, also the input actions have formal variables rather than values.

describe a richer structure, the *history-dependent automata* (*HD-automata* in brief), which are obtained by allowing names to appear explicitly in states, transitions and labels. As we will see, it is convenient to assume that the names which appear in a state, a transition or a label of a HD-automaton are *local* names and do not have a global identity. In this way, for instance, a single state of the HD-automaton can be used to represent all the states of a system that differ just for a bijective renaming. In this way, however, each transition is required to represent explicitly the correspondences between the names of source, target and label.

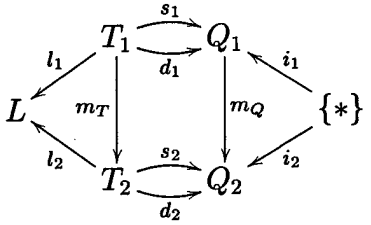
In this section we show that HD-automata can be defined in a categorical framework by extending the classical categorical definition of ordinary automata.

An ordinary automaton can be defined as a diagram

$$L \xleftarrow{l} T \underset{d}{\overset{s}{\rightrightarrows}} Q \xleftarrow{i} \{*\}$$

in the category **Set** of sets. Sets Q, T and L represent respectively the *states*, the *transitions* and the *labels* of the automaton. Functions s, d and l associate to each transition respectively its *source*, its *destination* state and its *label*. If $t \in T$ is such that $s(t) = q$, $d(t) = q'$ and $l(t) = \lambda$, then we write in brief $t : q \xrightarrow{\lambda} q'$. The initial state of the automaton is designated by $i(*)$.

Given two automata A_1 and A_2 on the same set L of labels, a *morphism* $m : A_1 \to A_2$ is a pair of arrows $m_Q : Q_1 \to Q_2$ and $m_T : T_1 \to T_2$ that respect sources, destinations, labels, and initial state, i.e., such that the two overlapped diagrams



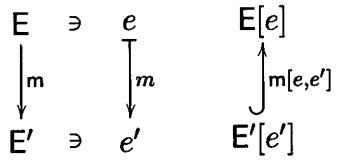
commute in the obvious way.

The category $\mathbf{Aut}_L$ of the automata on labels L is defined by using automata with labels L as objects and morphisms between such automata as arrows; identity arrow and composition between arrows are defined in the obvious way.

HD-automata can be defined in a similar way: we have just to replace the category **Set** with a category of *named sets*.

Definition 3.1 (named sets) A *named set* E is a set denoted by E, and a family of name sets indexed by E, namely $\{\mathsf{E}[e] \in Set\}_{e \in E}$ (i.e., $\mathsf{E}[\_]$ is a map form E to Set).
Given two named sets E and E', a *named function* $\mathsf{m} : \mathsf{E} \to \mathsf{E}'$ is a function on the sets $m : E \to E'$ and a family of name embeddings (i.e., of injective

functions) indexed by m, namely $\{\mathsf{m}[e,e'] : \mathsf{E}'[e'] \hookrightarrow \mathsf{E}[e]\}_{\langle e,e'\rangle \in m}$.

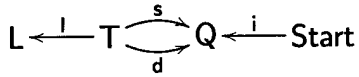


A named set E is *finitely named* if $\mathsf{E}[e]$ is finite for each $e \in E$. A named set E is *finite* if it is finitely named and set E is finite.

The *category* **NSet** *of named sets* has named sets as objects and named functions as arrows; in particular:

- if E is a named set, then id_E is the named function such that, for each $e \in E$, $id_E(e) = e$ and $\mathsf{id}_\mathsf{E}[e,e] = \mathrm{id}_{\mathsf{E}[e]}$;
- if $\mathsf{m} : \mathsf{E}_1 \to \mathsf{E}_2$ and $\mathsf{m}' : \mathsf{E}_2 \to \mathsf{E}_3$ are two named functions, then $\mathsf{m};\mathsf{m}' : \mathsf{E}_1 \to \mathsf{E}_3$ is the named function such that, for each $e \in E_1$, $m;m'(e) = m'(m(e))$ and, if $m(e) = e'$ and $m'(e') = e''$ then $\mathsf{m};\mathsf{m}'[e,e''] = \mathsf{m}'[e',e''];\mathsf{m}[e,e']$.

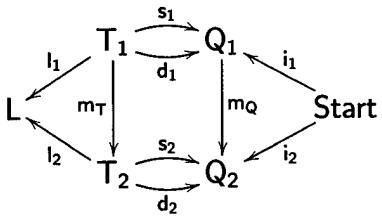
Definition 3.2 (HD-automata) Let Start be the named set with $*$ as singleton element and $\mathsf{Start}[*] = \mathfrak{N}$. A *HD-automaton* is a diagram



in the category **NSet** of named sets.

A HD-automaton is *finitely named* if L, Q and T are finitely named; it is *finite* if, in addition, Q and T are finite.

Given two HD-automata $\mathcal{A}_1$ and $\mathcal{A}_2$ on the same named set L of labels, a *morphism* $m : \mathcal{A}_1 \to \mathcal{A}_2$ is a pair of arrows $\mathsf{m}_Q : \mathsf{Q}_1 \to \mathsf{Q}_2$ and $\mathsf{m}_T : \mathsf{T}_1 \to \mathsf{T}_2$ that respects sources, destinations, labels, and initial state, i.e., such that the two overlapped diagrams



commute in the obvious way.

The category $\mathbf{HD}_\mathsf{L}$ of the HD-automata on labels L is defined as the full subcategory of **HD** whose objects have L as the set of labels and respect the following condition:

$\mathsf{T}[t] = \mathrm{cod}(\mathsf{s}[t, s(t)]) \cup \mathrm{cod}(\mathsf{l}[t, l(t)])$ for each $t \in T$.

Let t be a transition of a HD-automaton such that $s(t) = q$, $d(t) = q'$ and $l(t) = \lambda$ (in this case we write in brief $t : q \xrightarrow{\lambda} q'$). Then $\mathsf{s}[t,q]$ embeds the names of q into the names of t, whereas $\mathsf{d}[t,q']$ embeds the names of q' into the names of t; in this way, a partial correspondence is defined between the names

of the source state and those of the target; the names which appear in the source and not in the target are discarded, or forgotten, during the transitions, whereas the names that appear in the target but not in the source are created during the transition. Condition "$\mathsf{T}[t] = \mathrm{cod}(\mathsf{s}[t, s(t)]) \cup \mathrm{cod}(\mathsf{l}[t, l(t)])$" corresponds to require that all the names that are created in the transition must appear explicitly in the label (name discarding, instead, can appear silently).

The *initial state* q_0 of a HD-automaton is designated by $i(*)$, whereas $\mathsf{i}[*, q_0]$ is the *initial embedding* that maps the names of the initial state into the set $\mathfrak{N}$ of global names.

3.1 Well-sorted HD-automata

The HD-automata we have defined above are satisfactory for representing the operational semantics of many history dependent formalisms, like CCS with localities [20] and Petri nets with history-preserving bisimulation [19]. They are not completely adequate for the π-calculus.

In fact, let $p(a, b, c)$ and $q(a, b)$ be equivalent π-calculus agents with different sets of free names [3] and suppose the two agents perform a bound output. According to Definition 2.1, in checking bisimilarity we require that the object of the bound output is a new name for *both* agents. On the HD-automata this can be achieved by representing the bound output with an unique transition that introduces a new name.

If the two agents perform an input, however, all the names must be considered as possible input values. To represent the input on a HD-automaton, we have to consider a transition for each of the names which are present in the source state, and a transition corresponding to the input of a fresh name. In both the HD-automata corresponding to p and q, hence, there are transitions corresponding to the input of names a and b and to the input of a fresh name. The transition for name c appears only in the HD-automaton of p, since c is not free in q: this transition of p is matched in q by the transition for the fresh name.

This shows that the objects of bound outputs and of inputs have different meanings: in the case of bound outputs they are *new* names, whereas the objects of inputs are either already present in the source state or *universal* names (i.e., they represent *all the other* names, including the names which are free only in the other agent). In the HD-automata, however, there is only one way to introduce fresh names in a transition; so we need to add a new component to the HD-automata to distinguish between bound-output-like transitions and fresh-input-like transitions. This new component, called *sorting*, allows to distinguish the names of a label that *must* appear in the source (the `old` names), those that *cannot* appear in the source (the `new` names) and those that *may* appear in the source (the `both` names).

[3] This can be easily obtained, for instance by getting $p(a, b, c) = q(a, b) + (\nu x)\,\bar{x}c.0$, where the component $(\nu x)\,\bar{x}c.0$ is deadlocked.

Definition 3.3 (well-sorted HD-automata) Let L be a named set of labels. A *sorting* Γ for L associates to each label $\lambda \in L$ a function $\Gamma_\lambda : \mathsf{L}[\lambda] \to \{\mathtt{new}, \mathtt{old}, \mathtt{both}\}$.
The category $\mathbf{HD}_{\mathsf{L},\Gamma}$ of the well-sorted HD-automata on labels L and sorting Γ is defined as the full subcategory of $\mathbf{HD}_{\mathsf{L}}$ whose objects respect following condition:

for each $t \in T$ and $n \in \mathsf{L}[l(t)]$, if $\mathsf{I}[t, l(t)](n) \in \mathrm{cod}(\mathsf{s}[t, s(t)])$ then $\Gamma_{l(t)}(n) \neq \mathtt{new}$ and if $\mathsf{I}[t, l(t)](n) \notin \mathrm{cod}(\mathsf{s}[t, s(t)])$ then $\Gamma_{l(t)}(n) \neq \mathtt{old}$.

According to the previous considerations, the names of a transition $t : q \xrightarrow{\lambda} q'$ are classified as follows:

- $\mathsf{T}[t]_{\mathtt{new}} = \{n \mid n' \in \mathsf{L}[\lambda], \Gamma_\lambda(n') = \mathtt{new}, \mathsf{I}[t, \lambda](n') = n\}$ are the new names of transition t, i.e., the names which correspond to names of the label of sort `new`;
- $\mathsf{T}[t]_{\mathtt{src}} = \mathrm{cod}(\mathsf{s}[t, q])$ are the names of transition t that are already present in the source state;
- $\mathsf{T}[t]_{\mathtt{univ}} = \mathsf{T}[t]_{\mathtt{both}} \setminus \mathsf{T}[t]_{\mathtt{src}}$ are the *universal names* of transition t, i.e., the names which correspond to names of the label of sort `both` and which are not present in the source state.

Notice that $\mathsf{T}[t]_{\mathtt{src}}$, $\mathsf{T}[t]_{\mathtt{new}}$ and $\mathsf{T}[t]_{\mathtt{univ}}$ are a partition of $\mathsf{T}[t]$, i.e., they are disjoint and their union contains all the names of t.

4 Representing π-calculus agents as HD-automata

We are interested in the representation of π-calculus agents as HD-automata. First we define the named set of labels L_π for this language: we have to distinguish between synchronizations, inputs, free outputs and bound outputs. Thus the set of labels is

$$L_\pi = \{\mathtt{tau}, \mathtt{in}, \mathtt{in}_2, \mathtt{out}, \mathtt{out}_2, \mathtt{bout}\}$$

where $\mathtt{in}_2$ and $\mathtt{out}_2$ are used when subject and object names of inputs or free outputs coincide (these special labels are necessary, since the function from the names associated to a label into the names associated to a transition must be injective). No name is associated to `tau`, one name (n) is associated to $\mathtt{in}_2$ and $\mathtt{out}_2$ and two names (n_{sub} and n_{obj}) are associated to `in`, `out` and `bout`.

The sorting Γ_π on L_π is defined as follows:

λ	tau	in		$\mathtt{in}_2$	out		$\mathtt{out}_2$	bout	
$n \in L(\lambda)$	–	n_{sub}	n_{obj}	n	n_{sub}	n_{obj}	n	n_{sub}	n_{obj}
$\Gamma_\lambda(n)$	–	old	both	old	old	old	old	old	new

This means that the subject names of the labels must be old names, whereas the object names must be old in the case of free output, new in the case of

bound output and can be either old or new in the case of input.

To associate a HD-automaton to a π-calculus agent, we have to represent the derivatives of the agent as states of the automaton and their transitions as transitions in the HD-automaton; the names corresponding to a state are the free names of the corresponding agent, the names corresponding to a transition are the free names of the source state plus the new names (if any) appearing in the label of the transition. A label of L_π is associated to each transition in the obvious way.

This naive construction can be improved to obtain more compact HD-automata. Consider the agent $p = x(z).q(x,y,z)$; it can perform an infinite number of input transitions, corresponding to different received names. In the context of HD-automata, however, due to the local nature of names, the transitions of p corresponding to the input of all the names different from x and y are indistinguishable; so it is sufficient to consider just three input transitions for p, i.e., the inputs of names x and y, and the input of one representative of the fresh names.

Similarly, it is sufficient to consider just one bound output, whose extruded name is the representative of the names not appearing in the agent; finally, all the τ and the free output transitions have to be considered.

According to the following definition we choose to use the first name which does not appear free in p — namely $\min(\mathfrak{N} \setminus \mathtt{fn}(p))$ — as representative for the input and bound output transitions of p.

Definition 4.1 (representative transitions) A π-calculus transition $t : p \xrightarrow{\alpha} q$ is a *representative transition* if $\mathtt{n}(\alpha) \subseteq \mathtt{fn}(p) \cup \{\min(\mathfrak{N} \setminus \mathtt{fn}(p))\}$.

The following lemma shows that the representative transitions express, up to α-conversion, all the behaviors of an agent.

Lemma 4.2 *Let $t : p \xrightarrow{\alpha} q$, with $\alpha = ax$ (resp. $\alpha = \bar{a}(x)$), be a non-representative π-calculus transition. Then there is some representative transition $t' : p \xrightarrow{\alpha'} q'$, with $\alpha' = ay$ (resp. $\alpha' = \bar{a}(y)$), such that $q' = q\{y/x\ x/y\}$.*

If only representative transitions are used when building a HD-automaton from a π-calculus agent, the obtained HD-automaton is finite-branching (i.e., with a finite set of transitions from each state of the automaton).

Another advantage of using local names is that two agents differing only for a bijective substitution can be collapsed in the same state in the HD-automaton: we assume to have a function $\mathtt{norm}$ that, given an agent p, returns a pair $(q,\sigma) = \mathtt{norm}(p)$, where q is the representative of the class of agents differing from p for bijective substitutions and $\sigma : \mathtt{fn}(q) \to \mathtt{fn}(p)$ is the bijective substitution such that $p = q\sigma$.

Definition 4.3 (from π-calculus agents to HD-automata) The HD-automaton $\mathcal{A}_p$ corresponding to a π-calculus agent p is defined by the following

α	τ	xy		xx	$\bar{x}y$		$\bar{x}x$	$\bar{x}(y)$	
λ	tau	in		$\mathtt{in}_2$	out		$\mathtt{out}_2$	bout	
$n \in \mathsf{L}[\lambda]$	–	n_{sub}	n_{obj}	n	n_{sub}	n_{obj}	n	n_{sub}	n_{obj}
$\kappa(n)$	–	x	y	x	x	y	x	x	y

Table 2
Relations between π-calculus labels and labels of HD-automata.

rules:

- if $\mathtt{norm}(p) = (p', \sigma')$ then:
  - $p' \in Q$ is the initial state and $\mathsf{Q}[p'] = \mathtt{fn}(p')$;
  - σ' is the initial embedding;
- if $q \in Q$, $t : q \xrightarrow{\alpha} q'$ is a representative transition and $\mathtt{norm}(q') = (q'', \sigma)$, then:
  - $q'' \in Q$ and $\mathsf{Q}[q''] = \mathtt{fn}(q'')$;
  - $t \in T$ and $\mathsf{T}[t] = \mathtt{fn}(q) \cup \mathtt{bn}(\alpha)$;
  - $s(t) = q$, $d(t) = q''$, $\mathsf{s}[t, q] = \mathrm{id}_{\mathtt{fn}(q)}$ and $\mathsf{d}[t, q''] = \sigma$;
  - $l(t) = \lambda$ and $\mathsf{l}[t, \lambda] = \kappa$ are defined as in Table 2.

Lemma 4.4 *For every π-calculus agent p, the HD-automaton $\mathcal{A}_p$ is well-sorted for labels L_π and sorting Γ_π.*

For each π-calculus agent p, the HD-automaton $\mathcal{A}_p$ is obviously finitely named. Now we will identify a class of agents that generate a finite HD-automaton. This is the class of *finitary* π-calculus agents, which is defined like the corresponding class of CCS agents.

Definition 4.5 (finitary agents) The *degree of parallelism* $\mathtt{deg}(p)$ of a π-calculus agent p is defined as follows:

$$\mathtt{deg}(\mathbf{0}) = 0 \qquad \mathtt{deg}(\mu.p) = 1$$
$$\mathtt{deg}((\nu\alpha)\,p) = \mathtt{deg}(p) \qquad \mathtt{deg}(p|q) = \mathtt{deg}(p) + \mathtt{deg}(q)$$
$$\mathtt{deg}(p{+}q) = 1 \qquad \mathtt{deg}([x{=}y]p) = \mathtt{deg}(p)$$
$$\mathtt{deg}(A) = 1$$

A π-calculus agent p is *finitary* if $\max\{\mathtt{deg}(p') \mid p \xrightarrow{\mu_1} \cdots \xrightarrow{\mu_i} p'\} < \infty$.

Theorem 4.6 *Let p be a finitary π-calculus agent. Then the HD-automaton $\mathcal{A}_p$ is finite.*

An important class of finitary agents which can be characterized syntactically is the class of the agents with *finite control*, i.e., the agents without parallel composition in the body of recursive definitions. In this case, after an

initialization phase during which a finite set of processes acting in parallel is created, no new processes can be generated.

5 Bisimulation for HD-automata

In this section we introduce a notion of bisimulation on HD-automata and give some basic properties of this bisimulation. We also show that the definition of bisimulation on π-calculus agents is captured exactly by the bisimulation on HD-automata.

5.1 Open maps and bisimulations

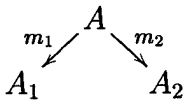
Consider a morphism $m : A_1 \to A_2$ in the category of automata. Relation

$$\mathcal{R} = \{\langle q_1, q_2\rangle \in Q_1 \times Q_2 \mid q_2 = m_Q(q_1)\}$$

is a simulation for A_1 and A_2. In fact, assume $q_1 \mathcal{R} q_2$ and $t_1 : q_1 \xrightarrow{\lambda} q_1'$; then we have $t_2 : q_2 \xrightarrow{\lambda} q_2'$ and $q_1' \mathcal{R} q_2'$ by taking $q_1' \mathcal{R} q_2'$. Moreover $q_{01} \mathcal{R} q_{02}$.

So, a morphism $m : A_1 \to A_2$ expresses the fact that all the transitions of A_1 can be simulated in A_2, starting from the initial states. In general, however, it is not true that all the transitions of A_2 can be simulated in A_1.

However, it is possible to define a particular class of "bisimulation" morphisms, such that the existence of such a morphism from A_1 to A_2 guarantees not only that the transitions of A_1 can be adequately simulated in A_2 but also the converse; i.e., the existence of a "bisimulation" morphism guarantees that A_1 and A_2 are bisimilar. In general, it is not true the converse, i.e., there exist bisimilar automata A_1 and A_2 such that no "bisimulation" morphism (nor generic morphisms) can be found between them. However, whenever two automata A_1 and A_2 are bisimilar, it is possible to find a common predecessor A and a span of "bisimulation" morphisms $m_1 : A \to A_1$ and $m_2 : A \to A_2$ between them.



This class of "bisimulation" morphisms have been defined in various manner in the literature, and different names have been given to them. Here we just consider the approach of *open maps* [10], that it is general enough to be applied not only to automata, but also to other models of concurrency, like Petri nets and event structures.

Assume a category $\mathbf{M}$ of *models*. Let $\mathbf{E}$ be the subcategory of $\mathbf{M}$ whose objects are the *experiments* that can be executed on $\mathbf{M}$ and whose arrows express how the experiments can be extended. If X is an object of $\mathbf{E}$ and M is an object of $\mathbf{M}$, an arrow $x : X \to M$ of $\mathbf{M}$ represents the execution of the experiment X in the model M.

Consider an arrow $m : M \to N$ in $\mathbf{M}$. We can see this arrow as a simulation of model M in model N. So, correctly, if an experiment X can be executed in M (there exists an arrow $x : X \to M$) and N can simulate M (there exists an arrow $m : M \to N$) then the experiment X can be executed in N (via the arrow $x; m : X \to N$).

Suppose now to extend the experiment X to an experiment Y (via an arrow $f : X \to Y$ in $\mathbf{E}$) and that an arrow $y : Y \to N$ exists such that the following diagram commutes in $\mathbf{M}$.

(1)
$$\begin{array}{ccc} X & \xrightarrow{x} & M \\ {\scriptstyle f}\downarrow & & \downarrow{\scriptstyle m} \\ Y & \xrightarrow{y} & N \end{array}$$

This means that the execution of the experiment X in N (via $x; m$) can be extended to an execution of the experiment Y in N (via y).

This does not imply in general that also the execution of X in M can be extended to an execution of Y in M (which equates y via m) but we can make this sure by requiring that there is an arrow y' such that the diagram

(2)
$$\begin{array}{ccc} X & \xrightarrow{x} & M \\ {\scriptstyle f}\downarrow & \overset{y'}{\nearrow} & \downarrow{\scriptstyle m} \\ Y & \xrightarrow{y} & N \end{array}$$

commutes. Given m, if for each commuting diagram (1) there is an arrow y' such that also (2) commutes, we say that m is an $\mathbf{E}$-*open map*.

It is easy to check that the open maps form a subcategory of $\mathbf{M}$ (i.e., identities are open and open maps are closed for composition).

Definition 5.1 (open bisimulation) We say that two objects M_1 and M_2 of $\mathbf{M}$ are *open-bisimilar* with respect to $\mathbf{E}$ if and only if there is a span of $\mathbf{E}$-open maps m_1, m_2.

$$\begin{array}{ccccc} & & M & & \\ & {\scriptstyle m_1}\swarrow & & \searrow{\scriptstyle m_2} & \\ M_1 & & & & M_2 \end{array}$$

In [10] it is shown that, if the category $\mathbf{Aut}_L$ is used as the category of the models and the full subcategory $\mathbf{Bran}_L$ of the *branches* (i.e., of those finite automata which consist of a linear sequence of transitions) is used as the category of experiments, then two automata are open-bisimilar if and only if they are bisimilar according to the classical definition.

5.2 Application to the HD-automata

In the case of HD-automata, an experiment is a finite sequences of transitions and an extended experiment can be obtained by adding new transitions. Moreover, we require that no name is forgotten during an experiment, since this models the idea that the observer can remember all the names previously

used in the experiment. However, this is not a crucial point for the validity of Theorem 5.4.

Definition 5.2 (category of HD-experiments) A HD-automaton $\mathcal{X}$ is a *HD-experiment* if:

- $Q = \{q_0, q_1, \ldots, q_n\}$ are the states and $T = \{t_1, \ldots, t_n\}$ are the transitions, and $s(t_i) = q_{i-1}$ and $d(t_i) = q_i$;
- for all $t \in T$, $\mathsf{d}[t, d(t)] : \mathsf{Q}[d(t)] \hookrightarrow \mathsf{T}[t]$ is bijective.

A morphism $\langle \mathsf{m_Q}, \mathsf{m_T} \rangle : \mathcal{X} \to \mathcal{X}'$ is *name preserving* if $\mathsf{m_Q}$ and $\mathsf{m_T}$ are bijections on the names, i.e., $\mathsf{m_Q}[q, m_Q(q)]$ is a bijection between $\mathsf{Q}'[m_Q(q)]$ and $\mathsf{Q}[q]$ for all $q \in Q$, and similarly for $\mathsf{m_T}$.
The *category* **Exp** *of HD-experiments* is the subcategory of **HD** with HD-experiments as objects and name preserving morphisms as arrows.
Category $\mathbf{Exp}_{\mathsf{L},\Gamma}$ is the full subcategory of **Exp** whose objects are $\mathbf{HD}_{\mathsf{L},\Gamma}$-automata.

Now we can apply the general definition of open-bisimilarity in our case.

Definition 5.3 (HD-bisimilarity) Two well-sorted HD-automata $\mathcal{A}$ and $\mathcal{B}$ on the same labels L and sorting Γ are *HD-bisimilar*, written $\mathcal{A} \sim \mathcal{B}$, if they are open-bisimilar w.r.t. experiments $\mathbf{Exp}_{\mathsf{L},\Gamma}$.

The definition of HD-bisimilarity can be applied also in the case of HD-automata obtained from π-calculus agents. The induced equivalence on the agents coincides exactly with the strong, early bisimilarity relation $\sim_\pi$.

Theorem 5.4 *Let p_1 and p_2 be π-calculus agents. Then $p_1 \sim_\pi p_2$ iff $\mathcal{A}_{p_1} \sim \mathcal{A}_{p_2}$.*

It is also possible to give an explicit definition of HD-bisimulation, in terms of relations on the states, rather that in terms of bisimulation morphisms. The explicit definition is reported in Appendix A.

6 Concluding remarks

In this paper we have briefly described history dependent automata, an operational model adequate to deal with history dependent calculi. In particular, we have represented π-calculus agents via HD-automata and strong, early π-calculus bisimilarity via a general definition of bisimulation equivalence on HD-automata. All these results will appear in more detail in [21].

We want to stress that HD-automata can be applied successfully also to the late semantics of π-calculus (only the translation of Definition 4.3 has to be changed) or to other examples of history dependent calculi, as for instance CCS with localities [4,20] or with causality [6,18]. It is also possible to define a *weak* HD-bisimulation that applies to all these cases. Also, HD-automata

can be applied to formalisms outside the field of process algebras; this is the case for the history-preserving semantics of Petri nets [2,19].

HD-automata are very promising for the development of automatic verification tools for history dependent calculi. In fact, HD-automata can be used as a common format in which various history-dependent calculi can be translated, so that general algorithms on HD-automata can be re-used for all these calculi. We are developing a verification environment which is based on the approach above. The environment provides a number of front ends translating the different history dependent formalisms into HD-automata, and a set of tools to edit, visualize, compose and check for equivalence the obtained HD-automata. It is also possible to associate ordinary automata to the HD-automata, in such a way that bisimilar HD-automata are mapped into bisimilar automata, and finite HD-automata are mapped into finite automata. In this way, classical algorithms and tools for ordinary automata [3] can be re-used. A preliminary report on the development of the tool appeared in [8].

References

[1] G. Berry and G. Boudol. The chemical abstract machine. *Theoretical Computer Science*, 96:217-248, 1992.

[2] E. Best, R. Devillers, A. Kiehn and L. Pomello. Fully concurrent bisimulation. *Acta Informatica*, 28:231–264, 1991.

[3] A. Bouali, S. Gnesi and S. Larosa. The integration project for the JACK environment. *Bullettin of the EATCS*, 54, 1994.

[4] G. Boudol, I. Castellani, M. Hennessy and A. Kiehn. Observing localities. *Theoretical Computer Science*, 114:31–61, 1993.

[5] M. Dam. On the decidability of process equivalences for the π-calculus. *Theoretical Computer Science*, 183:215-228, 1997.

[6] Ph. Darondeau and P. Degano. Causal trees. In *Proc. ICALP'89*, LNCS 372. Springer Verlag, 1989.

[7] P. Degano, R. De Nicola and U. Montanari. CCS is an (augmented) contact free C/E system. In *Proc. Adv. Sch. on Mathematical Models for the Semantics of Parallelism*, LNCS 280. Springer Verlag, 1986.

[8] G. Ferrari, G. Ferro, S. Gnesi, U. Montanari, M. Pistore and G. Ristori. An automata based verification environment for mobile processes. In *Proc. TACAS'97*, LNCS 1217. Springer Verlag, 1997.

[9] P. Inverardi and C. Priami. Evaluation of tools for the analysis of communicating systems. *Bulletin of the EATCS*, 45:158–185, 1991.

[10] A. Joyal, M. Nielsen and G. Winskel. Bisimulation from open maps. In *Proc. LICS'93*. Full version as BRICS RS-94-7. Department of Computer Science, University of Aarhus. 1994.

[11] A. Kiehn. Local and global causes. Tech. Rep. 42/23/91, Institut für Informatik, TU München, 1991.

[12] E. Madelaine. Verification tools for the CONCUR project. *Bulletin of the EATCS*, 47:110–126, 1992.

[13] R. Milner. *Communication and Concurrency.* Prentice Hall, 1989.

[14] R. Milner. The polyadic π-calculus: a tutorial. In *Logic and Algebra of Specification*, NATO ASI Series F, Vol. 94. Springer Verlag, 1993.

[15] R. Milner, J. Parrow and D. Walker. A calculus of mobile processes (parts I and II). *Information and Computation*, 100:1–77, 1992.

[16] R. Milner, J. Parrow and D. Walker. Modal logic for mobile processes. *Theoretical Computer Science*, 114:149-171, 1993.

[17] U. Montanari and M. Pistore. Checking bisimilarity for finitary π-calculus. In *Proc. CONCUR'95*, LNCS 962. Springer Verlag, 1995.

[18] U. Montanari and M. Pistore. History dependent verification for partial order systems. In *Partial Order Methods in Verification*, DIMACS Series, Vol. 29. American Mathematical Society, 1997.

[19] U. Montanari and M. Pistore. Minimal transition systems for history-preserving bisimulation. In *Proc. STACS'97*, LNCS 1200. Springer Verlag, 1997.

[20] U. Montanari, M. Pistore and D. Yankelevich. Efficient minimization up to location equivalence. In *Proc. ESOP'96*, LNCS 1058. Springer Verlag, 1996.

[21] U. Montanari and M. Pistore. History dependent automata. Tech. Rep. in preparation. Dipartimento di Informatica, Università di Pisa, 1998.

[22] D. Park. *Concurrency and Automata on Infinite Sequences*, LNCS 104. Springer Verlag, 1980.

[23] D. Sangiorgi. *Expressing Mobility in Process Algebras: First-Order and Higher-Order Paradigms.* PhD Thesis CST-99-93, University of Edinburgh, 1992.

[24] D. Walker. Objects in the π-calculus. *Information and Computation*, 116:253–271, 1995.

A Explicit definition of HD-bisimulation

Here we want to give an explicit definition of bisimulation on HD-automata which is equivalent to the one given in Definition 5.3 by exploiting the open maps. This definition is less satisfactory than the one given via open maps, since, as we will see, it has to deal explicitly with the different sorts of names that appear in a transition. However, the explicit definition makes it clear that the bisimulation of two HD-automata can be effectively decided whenever the two HD-automata are finite.

Due to the private nature of the names appearing in the states of HD-automata, bisimulations cannot simply be relations on the states; they must also deal with name correspondences: a HD-bisimulation is a set of triples of the form $\langle q_1, \delta, q_2 \rangle$ where q_1 and q_2 are states of the automata and δ is a partial bijection between the names of the states. The bijection is partial since we allow for equivalent states with different numbers of names (for instance, equivalent π-calculus agents can have different sets of free names). In what follows, we represent a *partial bijection* f from set A to set B with $f : A \rightharpoonup B$.

Suppose that we want to check if states q_1 and q_2 are (strongly) bisimilar via the partial bijection $\delta : \mathsf{Q}[q_1] \rightharpoonup \mathsf{Q}[q_2]$ and suppose that q_1 can perform a transition $t_1 : q_1 \xrightarrow{\lambda} q_1'$. We assume for the moment that all the names of the label λ are of sorts `old` or `new`. Then we have to find a transition $t_2 : q_2 \xrightarrow{\lambda} q_2'$ that matches t_1, i.e., not only the two transitions must have the same label, but also the names associated to the labels must be used consistently. This means that:

a) if a name n of the label is of sort `old`, then the corresponding names in the source states q_1 and q_2 must be in correspondence by δ (such names surely exist in q_1 and q_2, if the HD-automata are well-sorted);

b) if a name n of the label is of sort `new`, then the corresponding names in the transitions t_1 and t_2 are put in correspondence (if the HD-automata are well-sorted, no names corresponding to n appear in the source states).

This behavior is obtained by requiring that a partial bijection $\zeta : \mathsf{T}[t_1] \rightharpoonup \mathsf{T}[t_2]$ exists such that: *i*) ζ coincides with δ if restricted to the names of the source states (obviously, via the embeddings $\mathsf{s}[t_1, q_1]$ and $\mathsf{s}[t_2, q_2]$); *ii*) the names associated to the labels are the same, via ζ, and *iii*) the destination states q_1' and q_2' are bisimilar via a partial bijection δ' which is compatible with ζ (i.e., if two names are related by δ' in the destination states, then the corresponding names in the transitions are related by ζ).

The situation is more complex if a name n of the label λ is of sort `both`. We can distinguish three more cases:

c) the name n_1 in t_1 corresponding to n is already present in q_1 *and* is associated via δ' to a name of q_2; in this case a matching transition t_2 from q_2 must use for n this associated name;

d) the name n_1 in t_1 corresponding to n is already present in q_1 *and* it is *not* associated via δ to a name of q_2; in this case t_1 must be matched by a transition t_2 from q_2 which uses an universal name for n; the meaning of this is that name n_1 is handled as a special case in state q_1, but is handled by the default transition in q_2;

e) the name n_1 in t_1 corresponding to n is *not* already present in q_1 (i.e., n_1 is an universal name); in this case we require that t_1 is matched:

*e*1) by a transition from q_2 which uses an universal name for n (the two universal names are put in correspondence), and

e2) for each name n_2 of q_2 not appearing (via δ) in q_1, by a transition from q_2 that uses n_2 for n (in this case, a new correspondence is set for n_1 and n_2); the meaning of this is that the default transition in q_1 must match also the special cases of q_2 which are not contemplated by q_1.

This more complex behavior is obtained by requiring that, for each transition $t_1 : q_1 \xrightarrow{\lambda} q_1'$ and for each possible partial bijection ξ between the universal names of t_2 and the names of q_2 which do not already correspond to names of q_1, there is some transition $t_2 : q_2 \xrightarrow{\lambda} q_2'$ and some partial bijection $\zeta : \mathsf{T}[t_1] \rightharpoonup \mathsf{T}[t_2]$ extending $\xi; \mathsf{s}[t_2, q_2]$ such that *i*) ζ satisfies the rules *a-e*) above[4]; *ii*) the names associated to the labels are the same, via ζ, and *iii*) the destination states q_1' and q_2' are bisimilar via a partial bijection δ' which is compatible with ζ. Notice that to a name of t_1 can correspond no name of t_2 via ζ, if no name is associated to it via δ and the name does not appear in the label.

Definition A.1 (HD-bisimulation) Let $\mathcal{A}_1$ and $\mathcal{A}_2$ be two HD-automata in $\mathbf{HD}_{\mathsf{L},\Gamma}$. A *HD-simulation* for $\mathcal{A}_1$ and $\mathcal{A}_2$ is a set of triples $\mathcal{R} \subseteq \{\langle q_1, \delta, q_2\rangle \mid q_1 \in Q_1, q_2 \in Q_2, \delta : \mathsf{Q}_1[q_1] \rightharpoonup \mathsf{Q}_2[q_2]\}$ such that, whenever $\langle q_1, \delta, q_2\rangle \in \mathcal{R}$ then:

for each $t_1 : q_1 \xrightarrow{\lambda} q_1'$ in $\mathcal{A}_1$ and for each $\xi : \mathsf{T}_1[t_1]_{\mathtt{univ}} \rightharpoonup \mathsf{Q}_2[q_2]$ such that $\mathrm{cod}(\xi) \cap \mathrm{cod}(\delta) = \emptyset$, there exist some $t_2 : q_2 \xrightarrow{\lambda} q_2'$ in $\mathcal{A}_2$ and some $\zeta : \mathsf{T}_1[t_1] \rightharpoonup \mathsf{T}_2[t_2]$ such that:

- $\delta = \mathsf{s}_1[t_1, q_1]; \zeta; \mathsf{s}_2[t_2, q_2]^{-1}$;
- $\xi = \zeta|_{\mathsf{T}_1[t_1]_{\mathtt{univ}}}; \mathsf{s}_2[t_2, q_2]^{-1}$;
- $\mathsf{l}_1[t_1, \lambda]; \zeta = \mathsf{l}_2[t_2, \lambda]$;
- $\langle q_1', \delta', q_2'\rangle \in \mathcal{R}$ where $\delta' \subseteq \mathsf{d}_1[t_1, q_1']; \zeta; \mathsf{d}_2[t_2, q_2']^{-1}$.

A *HD-bisimulation* for $\mathcal{A}_1$ and $\mathcal{A}_2$ is a set of triples $\mathcal{R}$ such that $\mathcal{R}$ is a HD-simulation for $\mathcal{A}_1$ and $\mathcal{A}_2$ and $\mathcal{R}^{-1} = \{\langle q_2, \delta^{-1}, q_1\rangle \mid \langle q_1, \delta, q_2\rangle \in \mathcal{R}\}$ is a HD-simulations for $\mathcal{A}_2$ and $\mathcal{A}_1$.

A HD-bisimulation on for $\mathcal{A}$ is a HD-bisimulation for $\mathcal{A}$ and $\mathcal{A}$.

The HD-automata $\mathcal{A}_1$ and $\mathcal{A}_2$ are *(strongly) HD-bisimilar* (written $\mathcal{A}_1 \sim \mathcal{A}_2$) if there exists some HD-bisimulation for $\mathcal{A}_1$ and $\mathcal{A}_2$ such that $\langle i_1(*), \delta, i_2(*)\rangle \in \mathcal{R}$ for $\delta \subseteq \mathsf{i}_1[*, i_1(*)]; \mathsf{i}_1[*, i_1(*)]^{-1}$.

Theorem A.2 *Two HD-automata are bisimilar according to Definition 5.3 iff they are bisimilar according to Definition A.1.*

[4] For rule *e*): let n_1 be an universal name of t_1; if $\xi(n_1) = n_2 \in \mathsf{Q}[q_2]$ then n_1 must be matched by n_2 (case *e2*); if $\xi(n_1)$ is undefined, an universal name of t_2 must match n_1 (case *e1*).

Electronic Notes in Theoretical Computer Science 10 (1998)
URL: http://www.elsevier.nl/locate/entcs/volume10.html 8 pages

Can Actors and π-Agents Live Together?

Ugo Montanari [1]

Dipartimento di Informatica
University of Pisa, Italy
ugo@di.unipi.it

Carolyn Talcott [2]

Department of Computer Science
Stanford University
Stanford, CA, USA
clt@sail.stanford.edu

Abstract

The syntax and semantics of actors and π-agents is first defined separately, using a uniform, "unbiased" approach. New coordination primitives are then added to the union of the two calculi which allow actors and π-agents to cooperate.

1 Modeling Actors and Agents

The syntax and semantics of actors and π-agents are first defined separately, using a uniform, "unbiased" approach. Since we aim at modeling concurrent distributed systems, and thus we are interested in asynchronous behavior, we choose for comparison with actors an asynchronous version of the π-calculus.

In the paper, the behavior of both actors and π-agents is defined by certain logic sequents called *tiles*. A tile is a rewrite rule which describes a possible

[1] Research supported by Office of Naval Research Contracts N00014-95-C-0225 and N00014-96-C-0114, National Science Foundation Grant CCR-9633363, and by the Information Technology Promotion Agency, Japan, as part of the Industrial Science and Technology Frontier Program "New Models for Software Architechture" sponsored by NEDO (New Energy and Industrial Technology Development Organization). Also research supported in part by U.S. Army contract DABT63-96-C-0096 (DARPA); CNR Integrated Project *Metodi e Strumenti per la Progettazione e la Verifica di Sistemi Eterogenei Connessi mediante Reti di Comunicazione*; and Esprit Working Groups *CONFER2* and *COORDINA*. Research carried on in part while the first author was on leave at Computer Science Laboratory, SRI International, Menlo Park, USA, and visiting scholar at Stanford University.
[2] Research was partially supported by ONR grant N00014-94-1-0857, NSF grant CRR-9633419, and DARPA/Rome Labs grant AF F30602-96-1-0300,

evolution of a part s of the system which is matched by it. In addition, a tile also describes the evolution of the *interfaces* of s with the rest of the system. Thus two parts s and s' sharing an interface can be rewritten only by tiles which agree on the evolution of the common interface. This restriction introduces a powerful notion of synchronization [3] among tiles, and also makes possible to see the synchronization of two tiles as a (larger) tile. Eventually, all the possible evolutions are obtained by the repeated composition (synchronized, or in sequence, or in parallel) of certain small *basic* tiles called *rewrite rules.*

In the case of actors and π-agents, it is convenient to take a coordination [20] point of view and to distinguish between the behavior of agents in isolation and the behavior of *coordinators*, i.e. of system components whose role is to connect agents and to control their behavior. This approach allows us to abstract from the behavior *in the small* of agents, which is presented in a state transition, syntax-independent form, and to focus on the behavior of coordinators, which are the most characteristic feature of distributed systems. Correspondingly, we distinguish between two kinds of rewrite rules: *activity* rules and *coordination* rules. An activity rule describes an evolution of a single sequential agent, and may produce some action at its interface with the rest of the system. A coordination rule describes an evolution of a coordinator, and may both require certain actions from the agents it controls, and produce actions for a coordinator operating at a higher level.

For actors and π-agents, the interfaces with the rest of the system contain the free names of the agent, or, equivalently, the acquaintances (including self) of the actor. We call both of them names. An important difference with the ordinary semantics of both calculi is that in our approach names have only a local scope. Thus if in a particular subsystem there are n names, we can just denote them with $x_1, x_2, \ldots, x_n$. This choice makes the handling of names much easier, especially in the presence of bound (restricted) names and of name extrusion steps: it avoids α-conversion, infinite branching and in general the need of making provisions for an infinite number of possible names when connecting with the external world.

In addition to names, the interfaces contain also *events.* Events are the mechanism we use for establishing concurrency control in a distributed system. Agents and messages include references to the events which generated them, and to the previous events which caused these events. For instance in the *event diagram* semantics [12,34], when a message is received, a new event is created, and pointers to it and its causes are made available to all the components spawned by the step. The *causes* of this new event are the events both the message and the receiving agent pointed to. The causal relation determines the orderings in which the events can happen: all the sequential orderings compatible with the causal relation are possible, and they correspond to the

[3] Even in an asynchronous system, synchronization is required to model the reception of messages and the extrusion of names.

same *concurrent* computation.

In the representation of agents, actions and events, the key notion is *sharing*. In fact, the only role of a name is to specify which agents share it, and similarly several events may share the same cause. Sharing is a well-studied notion from a formal point of view, in particular for the subterms shared by a term. For instance, terms can be broken into the parallel and sequential composition of term constructors and basic substitutions, modulo certain axioms. Sharing in its purest form is then represented by the basic substitution ∇ from one variable to two values:

$$\nabla\colon 1 \to 2 = \{y_1 := x_1, y_2 := x_1\}.$$

However, in the algebra of terms and substitutions, sharing is not a first class component, in the sense that it can be freely removed by copying the shared subterm. In the representation of agents, instead of terms we use an extended version of *term graphs*. Term graphs are more expressive than terms, since terms graphs with a different amount of sharing among subterms are actually different. As a consequence, in term graphs, the ∇ operator becomes a basic constructor.

2 Interoperability of Actors and Agents

In the ordinary syntax of actors and π-agents, we have three configuration operators: parallel composition, restriction and renaming. Parallel composition is very powerful, since references to the same name on both operands are automatically identified. In our approach, since names are only local, parallel composition considers all names as different and yields the union of the two subsystems without establishing any connection between them. Names are actually identified by the Δ *matching* operator, which is thus our second coordinator. The Δ operator is analogous but opposite to ∇, since it merges two names, or two events, into one, rather that creating two instances of the same variable. It replaces both variable substitution $\_\,[\_\,/\,\_]$ and parallel composition $\_ \mid \_$, which turn out, somewhat surprisingly, to have analogous meanings.

Renaming, which is of difficult interpretation in a distributed setting, becomes useless and is discarded[4]. Restriction is mantained, essentially with the same meaning.

The main difference between actor calculus and π-calculus resides, in our setting, in the different typing of names and in the different versions of Δ's and their coordination rules. The free names of a π-agent are all typed c (for *channel*) and thus there is only one Δ, which we call Δ^c. Given an actor, its name is typed a, while its acquaintances are typed r (for *reference*). Also, all

[4] A weak notion of renaming, *permutations*, is used. They correspond exactly to substitutions of the form $\rho : 2 \to 2 = \{y_1 := x_2, y_2 := x_1\}$. However, they just describe a "wire twisting", they have no coordinating role, and no rewrite rule matches a substitution.

Coordinator	Δ^a	Δ^r	Δ^c	ν^{ca}	ν^{rc}
Type	$ar \to a$	$rr \to r$	$cc \to c$	$ca \to \epsilon$	$rc \to \epsilon$
Permeability to :					
output	n	y	y	n	n
input	y	$-$	y	n	n
synchronization	y	$-$	y	y	y

Table 1
Permeability of coordinators for name sharing.

the acquaintances of a message, including its addressee, are typed r. There are only two Δ's: Δ^r accepting two references and yielding a reference, and Δ^a accepting an actor and a reference and yielding an actor. There is no Δ accepting two actors: this restriction fully enforces the uniqueness of actor names.

The behaviour of a Δ is determined by its permeability to input/output actions and to synchronization. It is easy to see that Δ^c must be permeable to everything, while Δ^r cannot be presented with any input action, and thus cannot synchronize either. The most interesting case is Δ^a, which is permeable to input and synchronization, but impermeable to output. The rationale under this restriction is that a message cannot exit a system if its addressee is inside the system. The permeability of the various Δ coordinators is summarized in Table 1.

Actors and π-agents are connected by coordinators that match channels and actor names or references and, at the same time, restrict visibility of the matched pair. Coordinator ν^{ca} accepts a channel and an actor name and yields nothing: it allows a message to be sent from a π-agent to the named actor, by sending the message on the matching channel. Symmetrically, ν^{rc} accepts an actor reference and a channel and yields nothing: it allows a message to be sent from an actor to a π-agent receiving on the named channel, by sending the message to the matching actor reference. The hiding aspect of the actor-π coordinators enforces a clean separation between the two worlds, preserving the local behavior of individual agents and making the interaction invisible to the outside world. To enforce this separation, names communicated in messages are required to be newly created and the coordination rule matches and hides them appropriately. The permeability of the actor-π coordinators is also summarized in Table 1.

Both calculi are equipped with mobility, thus the amount of name sharing established at configuration time can be modified, actually only increased, at run time. In our setting, new Δ's are created only by input instantiation, by a synchronization where the output action is extruding (similar to a *Close* step for π-calculus) , or by an activity tile describing the forking of some actor or

some π-agent. In these cases only homogeneous subsystems, either both actors or both π-agents, are connected. Synchronization via an actor-π coordination rule creates a new coordinator to match (and hide) the actor and π versions of the communicated name.

For full composability we have required that agent behavior can not test for equality of names (no matching construct in process algebra terms). This restriction is also made for the actor language studied in [4]. The restriction was removed in the langauge studied in [24] in order to implement synchronous message passing (remote procedure call) in terms of asynchronous communication. An alternative might be to allow actors to receive on more than one port.

We conjecture that the partial order on events that is the observation of a tile-based computation for actors gives the same semantics to actor components as the interaction diagram semantics of [34] for the case of actor behaviors without matching. The event-based semantics for the π-calculus is new, and seems like a natural framework both for comparing calculi for concurrent/distributed computation and for semantic interoperation of heterogeneous systems.

While a presentation of the actual rewrite rules for actors, π-agents and their interaction would need a more technical presentation of the tile model and of the data structures we use, we hope that the above discussion gives some hints about our approach, its motivations and its advantages.

3 Related Work

The actor model [21,6,1,2] is one of the first and best known models for concurrent distributed systems and consists of independent computational agents which interact solely via asynchronous message passing. Semantic foundations for actor computation have been given in [4,32–34]. An approach to specifying and implementing mechanisms for coordination of actors based on reflection is described in [3].

The π-calculus [29] is one of the best studied examples of *mobile* process calculi, namely calculi in which the communication topology among processes can dynamically evolve when computation progresses. The asynchronous version of the π-calculus has been introduced in [8,22] and studied in [5].

The tile model, introduced in [17], is described in general terms in [18,19]. Tiles are much like SOS inference rules [31], but they can be composed horizontally, vertically and in parallel to build larger proof steps. Tile systems generalize Kim Larsen and Liu Xinxin *context systems* [23] since they allow for more general rule formats. The tile model also extends rewriting logic [25–27] (in the nonconditional case), since it takes into account rewritings with side effects and rewriting synchronization. Tile systems can be seen as double categories [14] and tiles themselves as double cells. They can be equipped with observational equivalences and congruences.

The combined used of tiles and term graphs [7,13] for modeling asynchronous π-calculus and CCS with locations [9] has been described in [15,16]. Also coordination models equipped with flexible synchronization primitives are presented in [30,11]. It is also possible [28,10] to translate the tile model into rewriting logic, in order to take advantage of important features of rewriting logic, like execution strategies and reflective logics, and to employ its existing implementations.

References

[1] G. Agha, *Actors: A Model of Concurrent Computation in Distributed Systems*, MIT Press 1986.

[2] G. Agha, *Concurent Object Oriented Programming*, CACM 33 (9), pp.125-141, 1990.

[3] G. Agha. Abstracting interaction patterns: A programming paradigm for open distribute systems. In E. Najm and J-B. Stefani, editors, *Formal Methods for Open Object-based Distributed Systems*, pages 135–153. Chapmn & Hall, 1997.

[4] G. Agha, I.A. Mason, S.F. Smith and C.L. Talcott, *A Foundation for Actor Computation*, J. of Functional Programming, 7, pp.1-72, 1997.

[5] R. Amadio, I. Castellani, D. Sangiorgi, *On Bisimulations for the Asynchronous π-calculus*, CONCUR'96, LNCS, 1996.

[6] Henry G. Baker and Carl Hewitt. Laws for communicating parallel processes. In *IFIP Congress*, pages 987–992. IFIP, August 1977.

[7] H.P. Barendregt, M.C.J.D. van Eekelrn, J.R.W. Glauert, J.R. Kennaway, M.J. Plasmeijer, M.R. Sleep, *Term Graph Reduction*, Proc. PARLE, Springer LNCS 259, 141–158, 1987.

[8] G. Boudol, Asynchrony and the π-calculus (note), *Rapport de Recherche 1702*, INRIA Sophia-Antipolis, May 1992.

[9] G. Boudol, I. Castellani, M. Hennessy and A. Kiehn, *Observing Localities*, Theoretical Computer Science, 114: 31–61, 1993.

[10] R. Bruni, J. Meseguer and U. Montanari, *Process and Term Tile Logic*, Technical Report, SRI International, to appear.

[11] R. Bruni and U. Montanari, *Zero-Safe Nets, or Transition Synchronization Made Simple*, in: Catuscia Palamidessi, Joachim Parrow, Eds, EXPRESS'97, ENTCS, Vol. 7, `http://www.elsevier.nl/locate/entcs/volume7.html`.

[12] W. D. Clinger. *Foundations of Actor Semantics.* PhD thesis, MIT, 1981. MIT Artificial Intelligence Laboratory AI-TR-633.

[13] A. Corradini, F. Gadducci, *A 2-Categorical Presentation of Term Graph Rewriting*, Proc. CTCS'97, Springer LNCS, to appear, 1997.

[14] C. Ehresmann, *Catégories Structurées*: I and II, Ann. Éc. Norm. Sup. 80, Paris (1963), 349-426; III, Topo. et Géo. diff. V, Paris (1963).

[15] G.L. Ferrari and U. Montanari, *A Tile-Based Coordination View of Pi-Calculus*, in: Igor Privara, Peter Ruzicka, Eds., Mathematical Foundations of Computer Science 1997, Springer LNCS 1295, 1997, pp. 52-70.

[16] G. Ferrari, U. Montanari, *Tiles for Concurrent and Located Calculi*, in: Catuscia Palamidessi, Joachim Parrow, Eds, EXPRESS'97, ENTCS, Vol. 7, `http://www.elsevier.nl/locate/entcs/volume7.html`.

[17] F. Gadducci, *On the Algebraic Approach to Concurrent Term Rewriting*, PhD Thesis, Università di Pisa, Pisa. Technical Report TD-96-02, Department of Computer Science, University of Pisa, 1996.

[18] F. Gadducci, U. Montanari, *The Tile Model*, in: Gordon Plotkin, Colin Stirling, and Mads Tofte, Eds., Proof, Language and Interaction: Essays in Honour of Robin Milner, MIT Press, to appear. Paper available from `http://www.csl.sri.com/ ugo/festschrift.ps`.

[19] F. Gadducci and U. Montanari, *Tiles, Rewriting Rules and CCS*, in: *J. Meseguer, Ed., Procs. Rewriting Logic and Applications*, First International Workshop, ENTCS, Vol. 4 (1996), `http://www.elsevier.nl/locate/entcs/volume4.html`.

[20] D. Garlan, D. Le Métayer, Eds., *Coordination Languages and Models*, LNCS 1282, 1997.

[21] C. Hewitt, *Viewing Control Structures as Patterns of Passing Messages*, J. of Artificial Intelligence, 8(3), pp.323-364, 1977.

[22] K. Honda, M. Tokoro, *An Object Calculus for Asynchronous Communication*, In: M. Tokoro, O. Nierstrasz, P. Wegner, Eds., Object-Based Concurrent Computing, Springer LNCS 612, 21–51, 1992.

[23] K.G. Larsen and L. Xinxin, *Compositionality Through an Operational Semantics of Contexts*, in Proc. ICALP'90, LNCS 443, 1990, pp. 526-539.

[24] I.A. Mason and C.L.Talcott, *A Semantically Sound Actor Translation* in Proc. ICALP'97, LNCS 1256, 1997, pp. 369-378.

[25] J. Meseguer, *Conditional Rewriting Logic as a Unified Model of Concurrency*, Theoretical Computer Science **96**, 1992, pp. 73-155.

[26] J. Meseguer, *Rewriting Logic as a Semantic Framework for Concurrency: A Progress Report*, in: U. Montanari and V. Sassone, Eds., *CONCUR'96: Concurrency Theory*, Springer LNCS 1119, 1996, 331-372.

[27] J. Meseguer, Ed., *Procs. Rewriting Logic and Applications*, First International Workshop, ENTCS, Vol. 4 (1996), `http://www.elsevier.nl /locate/entcs/volume4.html`.

[28] J. Meseguer and U. Montanari, *Mapping Tile Logic into Rewriting Logic*, in: Francesco Parisi-Presicce, Ed., Proc. 12th WADT Workshop on Algebraic Development Techniques, Spinger LNCS 1376, 1998, to appear. Available from `http://www. csl.sri.com/ ugo/wadt.ps`.

[29] R. Milner, J. Parrow, D. Walker, *A Calculus of Mobile Processes* (parts I and II), Information and Computation, 100:1–77, 1992.

[30] U. Montanari, F. Rossi, *Graph Rewriting and Constraint Solving for Modelling Distributed Systems with Synchronization*, in: Paolo Ciancarini and Chris Hankin, Eds., Coordination Languages and Models, LNCS 1061, 1996, pp. 12-27. Full paper submitted for publication available from `http://www.csl.sri.com/ ugo/graphs.ps`.

[31] G. Plotkin, *A Structural Approach to Operational Semantics*, Technical Report DAIMI FN-19, Computer Science Department, Aarhus University, 1981.

[32] C.L.Talcott, *An Actor Rewriting Theory*, in: *J. Meseguer, Ed., Procs. Rewriting Logic and Applications*, First International Workshop, ENTCS, Vol. 4 (1996), `http://www.elsevier.nl/locate/entcs/volume4.html`.

[33] C. L. Talcott. Interaction semantics for components of distributed systems. In E. Najm and J-B. Stefani, editors, *1st IFIP Workshop on Formal Methods for Open Object-based Distributed Systems, FMOODS'96*, 1996. proceedings published in 1997 by Chapman & Hall.

[34] C. L. Talcott. Composable semantic models for actor theories. In T. Ito M. Abadi, editor, *Theoretical Aspects of Computer Science*, number 1281 in Lecture Notes in Computer Science, pages 321–364. Springer-Verlag, 1997.

Electronic Notes in Theoretical Computer Science 10 (1998)
URL: http://www.elsevier.nl/locate/entcs/volume10.html 26 pages

Specification Diagrams for Actor Systems

Scott F. Smith

Department of Computer Science
The Johns Hopkins University
scott@cs.jhu.edu

1 Introduction

We propose here a new form of graphical notation for specifying open distributed object systems. The primary design goal is to make a form of notation for defining message-passing behavior that is expressive, intuitively understandable, and that has a formal underlying semantics.

Specification diagrams are graphical structures. Many specification languages that have achieved widespread usage have a graphical foundation: engineers can understand and communicate more effectively by graphical means. Popular graphical specification languages include Universal Modelling Language (UML) and its predecessors [RJB98], and StateCharts [Har87]. UML is the now-standard set of object-oriented design notations; it includes several different forms of graphical specification notation. Our aim here is a language with similar intuitive advantage but significantly greater expressivity and formal underpinnings. The language is also designed to be useful throughout the development lifecycle, from an initial sketch of the overall architecture to detailed specifications of final components that may serve as documentation of critical aspects of their behavior. Its design was inspired by concepts from actor event diagrams [Cli81], process algebra [Mil80,Hoa85], and UML Sequence Diagrams [RJB98]. In this particular presentation the underlying communication assumptions we use are those taken from the actor model: object- and not channel-based naming, and asynchronous fair message passing.

1.1 Actor Concepts

We provide here a very brief overview of actor concepts; see [Agh86,AMST97,Tal97] for more complete descriptions.

Actors are distributed, object-based message passing entities. Since actors are object-based, they each have a unique *name*, and actors may dynamically create other actors. Individual actors independently compute in par-

[1] This work was partially supported by NSF grant CCR-9312433

allel, and actors only communicate by message passing. Messages are sent asynchronously, so the sender may continue computing immediately after a message send. Messages are of the form $a \triangleleft M$, indicating message M is sent to actor a. Officially, we have

Definition 1.1 (A, M, MP) *Define $a \in \mathbf{A}$ to be a fixed countably infinite set of actor names; $M \in \mathbf{M}$ to be a set of message expressions; and,*

$$mp \in \mathbf{MP} = \mathbf{A} \triangleleft \mathbf{M} \cup \mathbf{A} \triangleleft \mathbf{M} : \kappa$$

to be the set of message packets for $\kappa \in \mathbf{Key}$ a countable set of keys (e.g. $\mathbf{Key} = \mathbf{Nat}$). We will write $\mathbf{A} \triangleleft \mathbf{M}\{ : \kappa\}$ for a message packet that may or may not have a key. The keys serve a technical purpose which is described below.

All messages must eventually arrive at their destination, but with arbitrary delay. At this point they may be queued if the destination actor is busy. Additionally, individual actor computations must never starve. These two guarantees of progress are the fairness assumptions of actor computation. There is no programming language for actors; one possible language is defined in [MT97] but others are possible. A fixed semantic framework for actors has been developed [AMST97,Tal97]. We will use this framework as the basis for the developments here.

Actor systems are intended to model open distributed computation. This means that the whole system will not be present, and the framework must assume some external actors are interacting with the local system. Additionally, of the local actors, only some of their names may be known by external entities; these are the receptionists. The external actors are notated as the set χ, and the receptionists the set ρ. These sets may grow over time: the external actors will grow based on names received in messages from the outside, and receptionists may grow if new local names are sent out in messages.

Actor systems may be modeled by the set of possible sequences of inputs and outputs they may perform over time. We call one such sequence, possibly infinite, an *interaction path*, and model actor system behavior by a set of such paths. The technical details of this model are summarized in section 3.

2 Specification Diagram Notation

We begin with the graphical notation and an informal idea of its meaning. Figure 1 presents the basic diagram components. Vertical lines indicate progress in time going down, expressing abstract causal ordering on events, with events above necessarily leading to events below. This causal ordering will be termed a *causal thread.* Note there is no necessary connection between these "threads" and actors or processes, they exist only at the semantic level: a single thread of causality may be multiple actors, and a single actor may have multiple threads of causality. The components listed in Figure 1 are composed to form

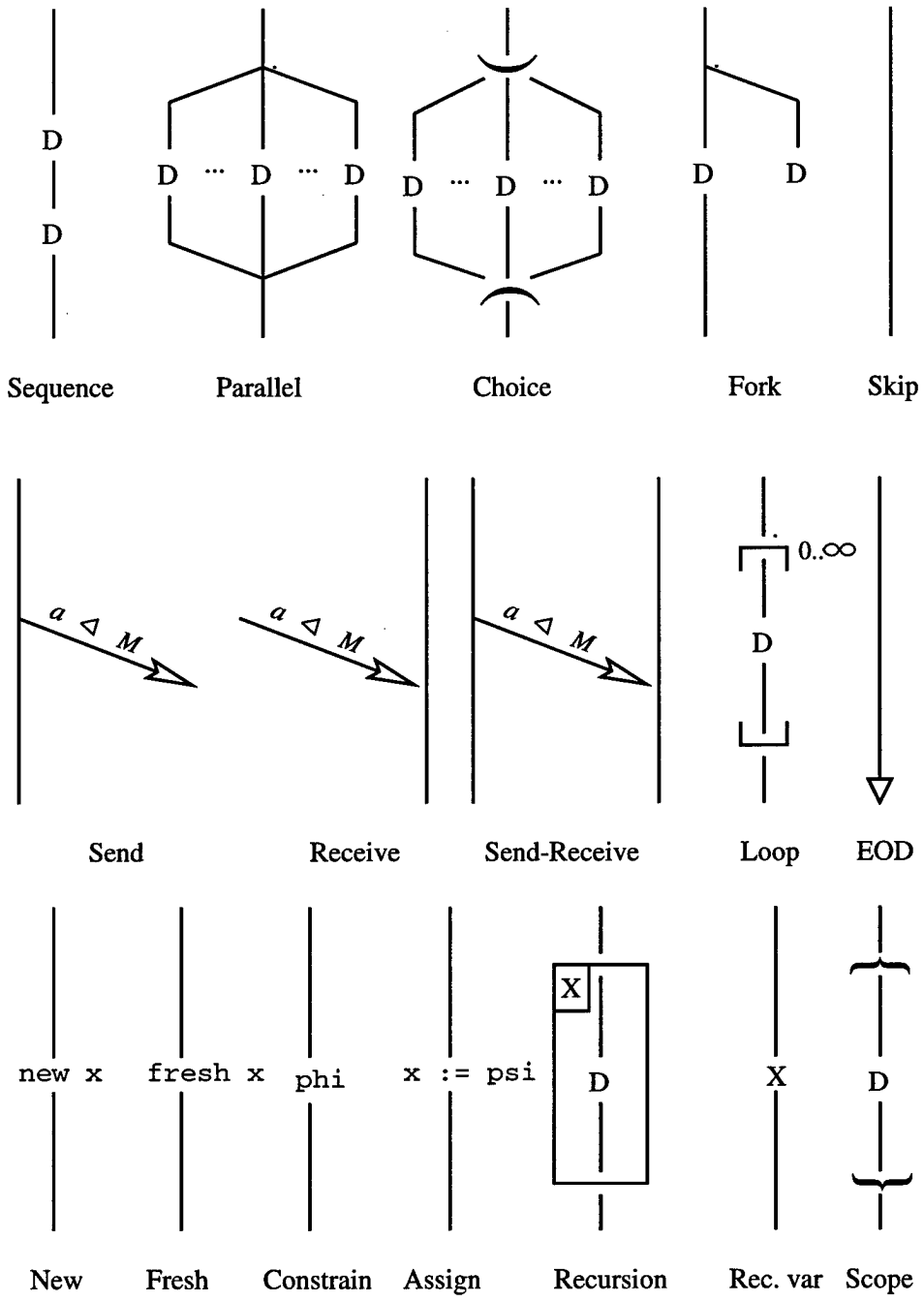


Fig. 1. Specification Diagram Components

specification diagrams. Each form of diagram component has a single in-edge and out-edge—the figure serves as a grammar for the diagrams.

sequencing Vertical lines (causal threads) represent necessary temporal sequencing of events.

parallel Events in the parallel diagrams have no causal ordering between them, but are after events above and before events below.

choice One of the possible choices is taken.

fork A diagram is forked off which hereafter will have no direct causal connection to the current thread (however, messages could indirectly impose some causality between the two).

skip Nothing.

send A message is sent.

receive A message is received by actor a, possibly binding pattern variables in the message M, which can be used below in the diagram.

send-receive A message is sent from one component of the diagram to another, producing a causal cross-connection in what could have been causally unrelated segments.

loop The diagram is iterated some number n times, where n is nondeterministically chosen from the interval $0 \ldots \infty$. The case $n = \infty$ means it loops forever.

EOD Denotes the end of a causal thred in the diagram.

scope Brackets demarcate static scoping of diagram variables.

recursion A boxed diagram fragment, may refer to itself by name, X.

new A fresh diagram variable x is allocated, with arbitrary contents.

fresh A fresh diagram variable x is allocated, with contents an actor name not currently in use.

constraint An arbitrary constraint is placed on the current thread of causality, which must be met. There is no direct analogue to this notion of constraint in programming languages: the constraint may be any mathematical expression. And, a constraint failing does not indicate an error, it indicates that such a computation path will not arise. So, it is significantly different from Hoare-style program assertions.

assign A variable is dynamically assigned a new value. The assignment body, ψ, can be any sensible mathematical expression.

Specification diagrams are truly a *specification* language—for internal messages, both source and target are shown, meaning there is a nonlocal constraint on both sender and receiver of message delivery. On this point they differ from existing forms of concurrent language.

There is no requirement that the choice be fair, in the sense that for a particular actor computation the same branch could always be taken. However, there is a requirement that message delivery be fair, in the sense that any message sent must eventually arrive at its destination.

The parallel and fork operators are similar, but parallel threads must eventually merge, while forked threads are asymmetrical in that the forked threads need never merge.

A *top-level* specification diagram includes an interface, notated D^{ρ}_{χ}. Top-level diagrams are modules which may be directly given semantic meaning. We will not always include the phrase "top level" but meaning should be clear from context.

In this brief paper we will use a textual notation and not the diagrammatic notation.

2.1 Textual Notation for Diagrams

We give an equivalent textual form for diagrams. This will also be considered the "official" syntax, and full details will be given. Diagrams may be easily mapped into the textual syntax.

The set of variables $\mathbf{X_d}$ is the set of diagram variables x, y, z, ... used in diagrams. These variables may take on values in some mathematical universe $\mathbf{U}$ which we keep open-ended; it could be instantiated with a set theory or type theory. We assume the basic collections of the actor theory are contained in $\mathbf{U}$: $\mathbf{A}, \mathbf{M}, \mathbf{MP}, \mathit{true}, \mathit{false} \subseteq \mathbf{U}$. To avoid circular mathematical definitions, we assume $\mathbf{U}$ is fixed before meaning of specification diagrams is assigned. And, we assume $acq(s)$ for $s \in \mathbf{U}$ is defined and constrain it to be a finite subset of $\mathbf{A}$.

The notion of message packet $\mathbf{MP}$ defined earlier included an optional *key*. Keys are necessary to give a textual representation to the send-receive edges in diagrams, constraining a send to be connected to its corresponding receive by sharing the same key. The interaction paths will not contain keys, they are used for local synchronization only.

Specification diagram message packets $\mathbf{MP_d}$ are packets of $\mathbf{MP}$ parameterized by diagram variables $\mathbf{X_d}$.

Definition 2.1 $(\mathbf{MP_d}, \mathbf{M_d}, \mathbf{A_d})$ *Let* $\mathbf{MP_d}$ *be* $\mathbf{A_d} \triangleleft \mathbf{M_d} \cup \mathbf{A_d} \triangleleft \mathbf{M_d} : \kappa$ *where* $\mathbf{A_d} = (\mathbf{X_d} \to \mathbf{U}) \to \mathbf{A}$ *and* $\mathbf{M_d} = (\mathbf{X_d} \to \mathbf{U}) \to \mathbf{M}$.

The parameter $\varepsilon \in \mathbf{X_d} \to \mathbf{U}$ is an environment interpreting free diagram variables. We use the informal convention that message expression $M_d = x+1$ is an abbreviation for $M_d(\varepsilon) = \varepsilon(x) + 1$.

Specification diagrams may now be defined officially by the following grammar.

Definition 2.2 (Specification Diagram Grammar) *The specification di-*

agrams, $D \in \mathbb{D}$, are defined by the following grammar.

name	*textual notation*
skip	`skip`
sequence	$D_1; D_2$
parallel	$D_1 \parallel D_2$
choice	$D_1 \oplus D_2$
scope	$\{D\}$
fork	$\texttt{fork}(D)$
receive	$\texttt{receive}(mp_{\mathrm{d}})$
send	$\texttt{send}(mp_{\mathrm{d}})$
constraint	$\texttt{constrain}(\phi)$
new variable	$\texttt{new}(x)$
fresh actor name	$\texttt{fresh}(x)$
assignment	$x := \psi$
recursion	$\texttt{rec}\, X.D$
recursion variable	X

where $D_i \in \mathbb{D}$ are diagrams, and $X \in \mathbf{X}_{\mathrm{r}}$ is a countable set of recursion variables. A top level *diagram is a diagram with an interface, D^{ρ}_{χ}.*

Sequencing is right-associative, and binds most tightly, followed by choice and then parallel composition binding most loosely. We use an implicit static variable binding convention: $\texttt{new}(x), \texttt{fresh}(x)$, and $\texttt{receive}(a_{\mathrm{d}} \triangleleft M_{\mathrm{d}}\{ : \kappa\})$, with x informally occurring in M_{d}, all denote implicit *bindings* (declarations) of variable x. $\{\ldots\}$ denotes a scope boundary, giving the static end of any implicit declarations contained immediately within. The extent of a variable binding is, intuitively, all points which causally must follow the binding, and that are at the same or deeper scope boundary. Thus for instance in $(\texttt{new}(x) \parallel \texttt{skip}); x := 5$, the assignment to x is within the scope of the `new` since this assignment causally must follow the declaration of x. If $\parallel$ were replaced by $\oplus$ in this example, the assignment x would not be bound (in fact it would be partially bound: if the left choice were taken it would be bound, but not if the right choice were taken).

The assignment expression ψ is informally a function on $\mathbf{U}$ which may refer to variables in $\mathbf{X}_{\mathrm{d}}$; formally, $\psi \in \mathbf{P}_{\omega}[\mathbf{A}] \times \mathbf{P}_{\omega}[\mathbf{A}] \times (\mathbf{X}_{\mathrm{d}} \to \mathbf{U}) \to \mathbf{U}$: it takes as parameter a tuple $(\rho, \chi, \varepsilon)$ where the environment ε gives mathematical meaning to the free variables, and ψ may refer to actors in

ρ and χ. Predicates on $\mathbf{U}$ are notated ϕ, and are formally predicates on $\mathbf{P}_\omega[\mathbf{A}] \times \mathbf{P}_\omega[\mathbf{A}] \times (\mathbf{X}_\mathrm{d} \to \mathbf{U})$. Informally, we will write e.g. "$x \in \rho$" to mean a predicate ϕ where $\phi(\rho, \chi, \varepsilon)$ iff $\varepsilon(x) \in \rho$. $acq(D)$ is defined as $\{acq(s) \mid s \in \mathbf{U}$ occurs as an mp_d, ϕ or ψ in $D\}$.

We use the convention that $\texttt{ExampleMacro}(x, y, z) = D$ defines a macro. Macros are just functions: we will be careful not to define self-referential macros. Certain syntax is easily encodable via macros and so is not defined in the core grammar.

Definition 2.3 (Diagram Macro Library) ***nondeterministic iteration:*** $[D]^{0\ldots\infty} = \texttt{rec}\, X.((D; X) \oplus \texttt{skip})$

interval iteration: $[D]^{i\ldots j} =$

$$\texttt{new}(x); \texttt{constrain}(i \leq x \leq j);$$
$$\texttt{rec}\, X.\texttt{constrain}(x = 0) \oplus \texttt{constrain}(x > 0); x := x - 1; D; X$$

for fresh variable x

if-then: $\texttt{if}\ \phi\ \texttt{then}\ D_1\ \texttt{else}\ D_2 =$

$$(\texttt{constrain}(\phi); D_1) \oplus (\texttt{constrain}(\neg\phi); D_2)$$

while-do: $\texttt{while}\ \phi\ \texttt{do}\ D =$

$$\texttt{rec}\, X.(\texttt{constrain}(\phi); D; X) \oplus \texttt{constrain}(\neg\phi)$$

end of diagram: $\texttt{eod} = \texttt{rec}\, X.(\texttt{skip}; X)$

abort path: $abort = \texttt{constrain}(false)$

initialized new: $\texttt{new}(x = s) = \texttt{new}(x); \texttt{constrain}(x = s)$

constrained new: $\texttt{new}(x \in S) = \texttt{new}(x); \texttt{constrain}(x \in S)$

constrained receipt: $\texttt{receive}(a_\mathrm{d} \triangleleft M_\mathrm{d} \in S) =$

$$\texttt{receive}(a_\mathrm{d} \triangleleft M_\mathrm{d}); \texttt{constrain}(M_\mathrm{d} \in S)$$

Translation from informal diagrams into textual notation is straightforward from the above in all cases except the internal message edges. With these internal edges, a diagram can take on an arbitrary graph structure, and this cannot be placed in purely textual notation. The purpose of keyed message packets is precisely to capture the 1-1 relation implied by an internal edge: the sender and receiver must be paired with no other send/receive to this message. Thus, for each internal edge in a diagram, a fresh key κ is assigned to it, and message $a_\mathrm{d} \triangleleft M_\mathrm{d}$ on the message edge translated into a $\texttt{send}(a_\mathrm{d} \triangleleft M_\mathrm{d} : \kappa)$ action by the sender and a $\texttt{receive}(a_\mathrm{d} \triangleleft M_\mathrm{d} : \kappa)$ action by the receiver.

2.2 Small Examples

We now give a series of small examples to illustrate use of the language. None of the examples attempts to seriously illustrate the usefulness of the language for specifications, as the examples are too small; the goal here is just to give informal clarification of the syntax and semantics.

Message Passing Semantics

Message passing in diagrams differs from actor programming. Diagram messages are specifications that a message was sent on one end and actively processed on the other. For instance, consider a sink:

$$\texttt{Sink}(a) = [\texttt{receive}(a \lhd x)]^{0\ldots\infty}$$

A sink behavior in an actor is a behavior that does nothing; messages will automatically queue up. At the specification level, we need to specify the receipt of those messages that are forever ignored. The diagrams do implicitly account for the asynchronous nature of actor communication: a message edge only constrains the send to be before the receive, not for the two to happen simultaneously. The use of "$0 \ldots \infty$" iteration models all possible environment behaviors. The environment may send 0, an arbitrary number, or infinitely many messages.

The interaction path semantics of the top-level diagram $\texttt{Sink}(a)^{\{a\}}_{\emptyset}$ have paths of the following form. Each path consists of a sequence of input messages $a \lhd M$. Since the system is reactive, the environment could send $0, 1, 2, \ldots, n, \ldots$ or even infinitely many such messages. No messages are sent out.

So, a point about message passing in diagrams is that messages which will *never* be processed must be specified as arriving. Consider

$$[\texttt{receive}(a \lhd x);\ \texttt{constrain}(x > 100)\ \ldots]^{0\ldots\infty}$$

This specification may at first look analogous to a synchronization constraint that only processes messages with contents larger than 100. However, a synchronization constraint of this form will implicitly forever ignore messages with contents less than 100. The above specification instead constrains the environment: it requires that such messages will in fact never be sent. Specifications which constrain the environment are *partial.*

Ticker

A ticker is a simple actor which increments a counter upon receipt of a `tick` message, and replies to `time` messages with the current time. First, a partial specification of a ticker is given which only specifies that time replies are numerical, not that the number of tick inputs is counted. This shows how

specifications may underconstrain the program space.

$$\begin{aligned}
&\texttt{PartialTicker}(a) = \\
&\quad [\texttt{receive}(a \triangleleft \texttt{tick})]^{0\cdots\infty} \parallel \\
&\quad [\texttt{receive}(a \triangleleft \texttt{time@}x);\ \texttt{new}(y \in \mathbf{Nat});\ \texttt{send}(x \triangleleft \texttt{reply}(y))]^{0\cdots\infty}
\end{aligned}$$

The possible interaction paths for the top-level diagram $\texttt{PartialTicker}\{a\}_{\emptyset}$ may be informally described as any path satisfying the following. For each $a \triangleleft \texttt{time@}c$ input in the path, there is later a $c \triangleleft \texttt{reply}(n)$ output for arbitrary $n \in \mathbf{Nat}$, and all outputs are `reply` messages so paired. There also may be any number, including infinitely many, $a \triangleleft \texttt{tick}$ input messages in any order.

Next we give a high-level specification of the full ticker: it gives a sequence of replies to `time` messages in a non-decreasing sequence.

$$\begin{aligned}
&\texttt{Ticker}(a) = \\
&\quad [\texttt{receive}(a \triangleleft \texttt{tick})]^{0\cdots\infty} \parallel \\
&\quad \texttt{new}(\texttt{count} \in \mathbf{Nat}); \\
&\quad [\texttt{receive}(a \triangleleft \texttt{time@}x);\ \texttt{new}(y \in \mathbf{Nat}); \\
&\quad \texttt{count} := \texttt{count} + y;\ \texttt{send}(x \triangleleft \texttt{reply}(\texttt{count}))]^{0\cdots\infty}
\end{aligned}$$

The actual implementation of a ticker we are specifying sends `tick` messages to itself to increment the counter: every time it receives a `tick`, it increments the counter and sends itself another `tick`. A low-level specification which is more close to an actual implementation is

$$\begin{aligned}
&\texttt{IntensionalTicker}(a) = \\
&\quad \texttt{send}(a \triangleleft \texttt{tick});\ \texttt{new}(\texttt{count} = 0); \\
&\quad [(\texttt{receive}(a \triangleleft \texttt{tick}); \\
&\quad \texttt{count} := \texttt{count} + 1;\ \texttt{send}(a \triangleleft \texttt{tick})) \\
&\quad \oplus\ \ (\texttt{receive}(a \triangleleft \texttt{time@}x);\ \texttt{send}(x \triangleleft \texttt{reply}(\texttt{count})))]^{0\cdots\infty}
\end{aligned}$$

Note that `tick` messages could in theory be sent by the environment. We then desire to establish equivalences on top-level diagrams such as

$$\texttt{Ticker}(a)_{\emptyset}^{\{a\}} \cong \texttt{IntensionalTicker}(a)_{\emptyset}^{\{a\}}$$

to show a high-level specification is equivalent to a low-level one. Extensional equivalence $\cong$ is defined in Section 5.

The above diagrams have a restricted acceptable message input set: messages must be in $\{\texttt{tick}, \texttt{time}\}$, and so the diagrams are partial. It is not difficult to extend a diagram to a diagram without acceptable message restrictions:

$$\texttt{CompleteTicker}(a) =$$
$$\texttt{fork}([\texttt{receive}(a \triangleleft x);\ \texttt{constrain}(\mathit{msgMeth}(x) \notin \{\texttt{tick}, \texttt{time}\})]^{0\ldots\infty});$$
$$\texttt{Ticker}(a)$$

Ticker Factory

This example shows how a fresh actor may be dynamically generated.

$$\texttt{TickerFactory}(a) =$$
$$[\texttt{receive}(a \triangleleft \texttt{new@}c);$$
$$\texttt{fresh}(x);$$
$$\texttt{fork}(\texttt{Ticker}(x));$$
$$\texttt{send}(c \triangleleft \texttt{reply}(x))]^{0\ldots\infty}$$

Function Composer

We assume as given two mathematical functions F and G which are defined over all possible messages. First we specify an abstract function-computer behavior.

$$\texttt{Function}(a, f) =$$
$$[\texttt{fork}(\texttt{receive}(a \triangleleft \texttt{compute}(x)\texttt{@}c);\ \texttt{send}(c \triangleleft \texttt{reply}(f(x))))]^{0\ldots\infty}$$

If the `fork` above was not present, the specification would implicitly order the function calls. However, since the function call order is irrelevant, the forkless specification would show causal ordering that was not necessary. The forking and forkless versions are in fact equivalent, however, in the sense that they specify the same set of interaction paths.

An actor which computes $G \circ F$ is then specified as $\texttt{Function}(a, G \circ F)$. A hypothetical implementation may use two distinct actors $\texttt{Function}(a_F, F)$ and $\texttt{Function}(a_G, G)$ to compute F and G respectively, and a `Composer` actor used to put together the composition. The system with these three actors should meet the above specification. The `Composer` is specified as follows.

$$\texttt{Composer}(a, f, g, a_f, a_g) =$$
$$[\texttt{receive}(a \triangleleft \texttt{compute}(x)@c);$$
$$\texttt{fresh}(x_f);\ \texttt{send}(a_f \triangleleft \texttt{compute}(x)@x_f);\ \texttt{receive}(x_f \triangleleft \texttt{reply}(x));$$
$$\texttt{fresh}(x_g);\ \texttt{send}(a_g \triangleleft \texttt{compute}(x)@x_g);\ \texttt{receive}(x_g \triangleleft \texttt{reply}(x));$$
$$\texttt{send}(c \triangleleft \texttt{reply}(x))]^{0\ldots\infty}$$

The complete low-level specification is then the parallel composition of these three:

$$\texttt{FunctionComposer}(a, F, G, a_F, a_G) =$$
$$\texttt{Composer}(a, F, G, a_F, a_G) \parallel \texttt{Function}(a_F, F) \parallel \texttt{Function}(a_G, G)$$

In an open distrubuted system, components may be designed as separate top-level diagrams and then composed. In particular for large systems for which the design is distributed across different political entities, object-based components are composed. For this example, the top-level composition is defined as follows.

$$\texttt{FunctionComposerModule}(a, F, G, a_F, a_G)^{\{a\}}_{\emptyset} =$$
$$\texttt{Composer}(a, F, G, a_F, a_G)^{\{a\}}_{\{a_F, a_G\}} \parallel \texttt{Function}(a_F, F)^{\{a_F\}}_{\emptyset} \parallel \texttt{Function}(a_G, G)^{\{a_G\}}_{\emptyset}$$

where $\parallel$ is not the composition operator on diagrams, but the composition operator on top-level specifications. Theorem 5.3 below may be used to show the two methods for composition produce the same specification:

$$\texttt{FunctionComposer}(a, F, G, a_F, a_G)^{\{a\}}_{\emptyset} \cong \texttt{FunctionComposerModule}(a, G \circ F)^{\{a\}}_{\emptyset}$$

This specification is equivalent to the abstract $\texttt{Function}(a, G \circ F)$:

$$\texttt{FunctionComposer}(a, F, G, a_F, a_G)^{\{a\}}_{\emptyset} \cong \texttt{Function}(a, G \circ F)^{\{a\}}_{\emptyset}$$

We sketch how an argument to establish this equivalence would proceed. The first step is to perform a "zipping" transformation on FunctionComposer to connect send and receive to give send-receive cross-edges. This is a point where the diagrammatic semantics becomes significantly more readable than

the textual form.

$$\begin{aligned}
&\texttt{ZippedFunctionComposer}(a, f, g, a_f, a_g) = \\
&\quad [\texttt{receive}(a \triangleleft \texttt{compute}(x)@c);\ \texttt{fresh}(x_f);\ \texttt{fresh}(x_g); \\
&\quad \texttt{send}(a_f \triangleleft \texttt{compute}(x)@x_f : \kappa_f);\ \texttt{receive}(x_f \triangleleft \texttt{reply}(f(x)) : \kappa'_f); \\
&\quad \texttt{send}(a_g \triangleleft \texttt{compute}(x)@x_g : \kappa_g);\ \texttt{receive}(x_g \triangleleft \texttt{reply}(g(x)) : \kappa'_g); \\
&\quad \texttt{send}(c \triangleleft \texttt{reply}(x))]^{0\ldots\infty} \parallel \\
&\quad [\texttt{fork}(\texttt{receive}(a_f \triangleleft \texttt{compute}(x)@x_f : \kappa_f);\ \texttt{send}(x_f \triangleleft \texttt{reply}(f(x) : \kappa'_f)))]^{0\ldots\infty} \parallel \\
&\quad [\texttt{fork}(\texttt{receive}(a_g \triangleleft \texttt{compute}(x)@x_g : \kappa_g);\ \texttt{send}(x_g \triangleleft \texttt{reply}(g(x) : \kappa'_g)))]^{0\ldots\infty}
\end{aligned}$$

The ZippedFunctionComposer is an intermediate stage in the incremental translation of FunctionComposer to Function. By proving

$$\begin{aligned}
&\texttt{FunctionComposer}(a, F, G, a_F, a_G)^{\{a\}}_{\emptyset} \cong \\
&\texttt{ZippedFunctionComposer}(a, F, G, a_F, a_G)^{\{a\}}_{\emptyset} \cong \\
&\texttt{Function}(a, G \circ F)^{\{a\}}_{\emptyset}
\end{aligned}$$

the desired equivalence is established.

Simple Memory Cell

$$\begin{aligned}
&\texttt{Cell}(a) = \\
&\quad \texttt{new}(x);\ \texttt{(* cell value, initially arbitrary *)} \\
&\quad [\ (\texttt{receive}(a \triangleleft \texttt{set}(v)@c);\ \texttt{(* c/v are pattern variables *)} \\
&\quad\ x := v; \\
&\quad\ \texttt{send}(c \triangleleft \texttt{ack})) \\
&\quad \oplus \\
&\quad\ (\texttt{receive}(a \triangleleft \texttt{get}@c); \\
&\quad\ \texttt{send}(c \triangleleft \texttt{reply}(x))\ ]^{0\ldots\infty}
\end{aligned}$$

A top-level specification for a memory cell is then $\texttt{Cell}(a)^{\{a\}}_{\emptyset}$. Another powerful idea is to write a partial specification which only responds to message input patterns that are semantically sensible. For instance, for a cell, receipt of a get before any set messages have arrived is unreasonable; we thus make

a specification which constrains the environment to never do this.

$\texttt{EnvironmentConstrainingCell}(a) =$

$\texttt{new}(x = \texttt{undefined})$; (* cell initially undefined *)

[$(\texttt{receive}(a \triangleleft \texttt{set}(v)@c)$;

$x := v$;

$\texttt{send}(c \triangleleft \texttt{ack})$)

$\oplus$

$(\texttt{receive}(a \triangleleft \texttt{get}@c); \texttt{constrain}(x \neq \texttt{undefined})$

$\texttt{send}(c \triangleleft \texttt{reply}(x))$ $]^{0..\infty}$

Environment constraints are a powerful aspect of specification diagrams. Such specifications are partial: they are not defined for all patterns of input. Composition with such specifications is also partial, as it may fail since the specification is not fully reactive. However, the concept of satisfaction of an environment-constrained specification is a difficult one and so now we will generally study complete specifications only.

3 A Path-Based Framework for Actors

In this section we briefly review the semantic framework used to model actor systems; this framework will then be used to model specification diagrams. See [Tal97] for details; here we provide a terse and simplified presentation. We use a path-based (trace-based) semantics: an open, nondeterministic system is interpreted as a set of *interaction paths*. Each path is a possibly infinite list of input and output actions. Interaction path semantics models an actor system in terms of the possible interactions (patterns of message passing) it can have with its environment. Interaction semantics does not let us use any information about internal computations or what actors may be initially present or known beyond those specificed in the interface. A specification diagram is given meaning as a set of interaction paths, so the meaning of a diagram D may be the same as the meaning of some actor program implementation. If this is in fact the case, we can assert that the implementation meets the specification D.

Interfaces

An actor system interface is a pair $\frac{\rho}{\chi}$ of disjoint finite sets of actor names. ρ specifies the receptionists and χ specifies the external actors known to the system.

We define parallel composition of interfaces by

$$\begin{smallmatrix}\rho_1\\ \chi_1\end{smallmatrix} \bowtie \begin{smallmatrix}\rho_2\\ \chi_2\end{smallmatrix} \quad \text{iff} \quad \rho_1 \cap \rho_2 = \emptyset$$

$$\begin{smallmatrix}\rho_1\\ \chi_1\end{smallmatrix} \parallel \begin{smallmatrix}\rho_2\\ \chi_2\end{smallmatrix} = \begin{smallmatrix}\rho_1 \cup \rho_2\\ (\chi_1 \cup \chi_2) - (\rho_1 \cup \rho_2)\end{smallmatrix}, \text{ provided } \begin{smallmatrix}\rho_1\\ \chi_1\end{smallmatrix} \bowtie \begin{smallmatrix}\rho_2\\ \chi_2\end{smallmatrix}$$

Events

In order to distinguish differrent occurrences of message packets with the same contents we use a set, $\mathbf{E}$, of events. Each event, e, contains a message packet, $pkt(e) \in \mathbf{MP}$. We let e range over $\mathbf{E}$ and E range over $\mathcal{P}[\mathbf{E}]$. We write $\mathtt{in}(e)$ to distinguish the arrival of a message from outside the system from its delivery e. We also write $\mathtt{out}(e)$ to distinguish delivery of a message to the environment from from the actual delivery to the target actor. We let $\widetilde{\mathbf{E}}$ be the set of events extended by these input/output events

$$\widetilde{\mathbf{E}} = \mathbf{E} \cup \mathtt{in}(\mathbf{E}) \cup \mathtt{out}(\mathbf{E})$$

and we let $\tilde{e}$ range over $\widetilde{\mathbf{E}}$ and $\widetilde{E}$ range over $\mathcal{P}[\widetilde{\mathbf{E}}]$. We will use a simple theory of potentially infinite lists, notated $[x_1, x_2, \ldots, x_k, \ldots]$. List concatenation is written $[\ldots] * [\ldots]$, and for the case where the first list is infinite returns that as the result. Unit for concatenation is the empty list, $[\,]$. S **List** is the set of lists with elements from S.

Interaction Paths

An interaction path is a possibly infinite list of input/output events,

$$\pi = [\tilde{e}_1, \tilde{e}_2, \ldots, \tilde{e}_k, \ldots] \in (\mathtt{in}(\mathbf{E}) \cup \mathtt{out}(\mathbf{E}))\ \mathbf{List}.$$

It represents a potentially infinite sequence of interactions of a system with its environment as observed by some hypothetical observer. π^{ρ}_{χ} represents an interfaced interaction path, a path with the indicated interface. All of the interaction paths constructed in this paper are constrained to obey the *EPLaw* of [Tal97]. *EPLaw*(π) requires inputs of π to be to receptionists or names sent in a previous output, and outputs to be to external actors or actors whose name was received in a previous input. *EPLaw* corresponds to the Baker-Hewitt locality laws governing how actors become acquainted with one another.

3.1 Interaction Path Sets and their Algebra

An interaction path models one possible way a system might interact with its environment. We model the behavior of a system by sets of interfaced interaction paths, *Ip*.

Parallel Composition

We define composability and composition on interaction path sets. The basic operation for composing paths is dovetailing two interaction paths, $\pi_0 \; \mathsf{Z} \; \pi_1$. This operation is defined in terms of precursor $\pi_0 \; \mathsf{Z}^0 \; \pi_1$, which is the greatest symmetric function closed under the following (in the following we abuse notation and write $[\ldots] * S$ to mean $\{[\ldots] * s \mid s \in S\}$).

$$(0)\ [\,]\ \mathsf{Z}^0\ \pi = \{\pi\}$$

$$(1)\ [\mathtt{in}(e)] * \pi_0\ \mathsf{Z}^0\ [\mathtt{out}(e)] * \pi_1 = \pi_0\ \mathsf{Z}^0\ \pi_1$$

$$(2)\ [\tilde{e}_0] * \pi_0\ \mathsf{Z}^0\ [\tilde{e}_1] * \pi_1 =$$
$$\{[\tilde{e}_0] * (\pi_0\ \mathsf{Z}^0\ [\tilde{e}_1] * \pi_1)\} \cup$$
$$\{[\tilde{e}_1] * ([\tilde{e}_0] * \pi_0\ \mathsf{Z}^0\ \pi_1)\}$$

Then, $\pi_0 \; \mathsf{Z} \; \pi_1$ is defined as $\pi_0 \; \mathsf{Z}^0 \; \pi_1$ with the paths $\pi \in \pi_0 \; \mathsf{Z}^0 \; \pi_1$ that after some point forever starve events from one of π_0 or π_1 removed.

Define composability and parallel composition for path sets Ip_0 and Ip_1 with interfaces $\frac{\rho_0}{\chi_0}$ and $\frac{\rho_1}{\chi_1}$ as

$$Ip_0 \bowtie Ip_1 \text{ iff } \tfrac{\rho_0}{\chi_0} \bowtie \tfrac{\rho_1}{\chi_1}$$

$$Ip_0 \parallel Ip_1 = \{\pi_\chi^\rho \mid (\exists\, \pi_0 {}_{\chi_0}^{\rho_0} \in Ip_0, \pi_1 {}_{\chi_1}^{\rho_1} \in Ip_1)$$
$$(\pi \in \pi_0\ \mathsf{Z}\ \pi_1,\ \pi_0 \text{ and } \pi_1 \text{ share no events } \tilde{e}, \text{ and } EPLaw(\pi))\}$$

where $\frac{\rho}{\chi} = \frac{\rho_0}{\chi_0} \parallel \frac{\rho_1}{\chi_1}$, provided $\frac{\rho_0}{\chi_0} \bowtie \frac{\rho_1}{\chi_1}$.

Restriction

The restriction of Ip with interface $\frac{\rho}{\chi}$ to ρ' is defined by

$$Ip\lceil\rho' = \{\pi_\chi^{\rho'} \mid \pi_\chi^\rho \in Ip \text{ and } \pi \text{ contains no } \mathtt{in}(a \triangleleft M) \text{ events for } a \in (\rho - \rho')\}$$

Renaming

Renaming of interaction paths, $\widehat{\alpha}(\pi)$, is pointwise on each event in the path, and renaming on path sets $\widehat{\alpha}(Ip)$ is pointwise on each element of the set.

4 Operational Semantics of Diagrams

Diagrams are given meaning in this section via an operational semantics. The goal is to give a set of interaction paths defining the behavior of top-level diagrams D_χ^ρ. This is accomplished by defining a single-step relation mapping configurations to configurations, the transitive closure of which yields a set of computation paths. In this sense it is a standard presentation of operational

semantics of actors [AMST97,Tal97]. The main difference with these previous works is more post-processing is required to remove paths that are not admissible. A configuration is of the form

$$D^{\rho}_{\chi}\,\mu$$

Where ρ, χ are the current receptionists and external actor names, and $\mu \subseteq \mathbf{E}$ is a set of messages in transit, either to be sent out of the system or to be received locally. Since each message is a unique event, it is possible to have two messages with identical target and contents in μ.

4.1 Preliminaries

Before presenting the operational semantics, we need to define two concepts. We use a small-step semantics based on factoring a diagram D expression into a redex D_{rdx} and reduction (a.k.a. evaluation) context R, $D = R[D_{\text{rdx}}]$. Notation is also needed for looking up, modifying, and extending variable bindings. The concepts of reduction context and environment are in fact intertwined: environments are local to particular points in reduction (so e.g. parallel threads may have differing environments) and so are spread around the reduction context. These local environments are functions $\gamma_i \in \mathbf{X}_{\text{d}} \to \mathbf{U}$ which hold the current state of diagram variables $\mathbf{X}_{\text{d}}$, and map only finitely many variables to non-$\perp$ values, $\perp \in \mathbf{U}$ being a special meta-value indicating undefined. A small addition to the language syntax is used to bind variables inside the executing diagram: $\{\!|\gamma : D|\!\}$ indicates a lexical scoping construct $\{D\}$ under which execution is actively occurring, with current local environment γ.

Reduction contexts R (a.k.a. evaluation contexts) are used to isolate the next redex to be reduced. Their grammar is

$$R = \bullet \text{ or } R \parallel D \text{ or } D \parallel R \text{ or } R; D \text{ or } \{\!|\gamma : R|\!\}$$

$R[D]$ denotes the act of replacing the (unique) $\bullet$ in R with diagram D. We will need notation for reduction contexts defined as above but without the $\{\!|\gamma : R|\!\}$ case; they will be notated R^{-}.

Notation is next defined for manipulation of the environment. The basic operations needed include $R@x$ to look up the value of x in the environments of R, $R@x := s$ to modify the value of already-declared variable x, and $R \oplus x := s$ to add a new definition of x in the innermost lexical level. $\varepsilon(R)$ extracts the environment from R in the form of a function from diagram variables to values.

Definition 4.1 ($R@x, R@x := s, R \oplus x := s$, $\varepsilon(R)$) *Letting*

$$R = R_0^{-}[\{\!|\gamma_1 : \ldots R_n^{-}[\{\!|\gamma_n : R_{n+1}^{-}|\!\}] \ldots |\!\}],$$

define

lookup $R@x$ *is* $\gamma_i(x)$ *where* i *is the largest number less than or equal to* n *with* $\gamma_i(x) \neq \bot$, *or* $\bot$ *if all* $\gamma_i(x) = \bot$.

modify $R@x := s$ *is* $R_0^-[\{\!|\gamma_1 : \ldots R_i^-[\{\!|\gamma_i' : \ldots R_n^-[\{\!|\gamma_n : R_{n+1}^-|\!\}] \ldots |\!\}] \ldots |\!\}]$ *where* i *is the largest number less than or equal to* n *with* $\gamma_i(x) \neq \bot$, *and* $\gamma_i' = \gamma_i$ *at all points except* $\gamma_i'(x) = s$. *If no such* i *exists,* R *is unchanged.*

extend $R \oplus x := s$ *is* $R_0^-[\{\!|\gamma_1 : \ldots R_n^-[\{\!|\gamma_n' : R_{n+1}^-|\!\}] \ldots |\!\}]$ *for* $\gamma_n' = \gamma_n$ *at all points except* $\gamma_n'(x) = s$.

extract $\varepsilon(R) = f$ *where* $f(x) = R@x$. $\mathrm{Dom}(\varepsilon(R)) = \{x \mid (\varepsilon(R))(x) \neq \bot\}$.

acquaintances $acq(D)$ *is*

$$\bigcup_{M_\mathrm{d} \in D} acq(M_\mathrm{d}) \cup \bigcup_{\phi \in D} acq(\phi) \cup \bigcup_{\psi \in D} acq(\psi) \cup \bigcup_{a_\mathrm{d} \in D} acq(a_\mathrm{d}) \cup \bigcup_{\gamma \in D} \bigcup_{x \in \mathbf{X}_\mathrm{d}} acq(\gamma(x))$$

where "$\cdot \in D$*" here means occurrence as a subterm in* D. $acq(R)$ *is defined identically to* $acq(D)$ *except with the last clause being* $\bigcup_{1 \leq i \leq n} \bigcup_{x \in \mathbf{X}_\mathrm{d}} acq(\gamma_n(x))$.

4.2 The Semantic Definition

In this section we define the semantic meaning function $[\![D_\chi^\rho]\!]$ mapping top-level diagrams to sets of interaction paths that describe the input-output behavior of the diagram.

Definition 4.2 *The single-step computation relation on diagram configurations is defined in Figure 2.*

For each of the rules except `in`/`out`, the *redex* is the D_rdx for left-hand-side $R[D_\mathrm{rdx}]$.

Definition 4.3 *Given a top-level diagram* $D_{0\chi_0}^{\rho_0}$, *define*

raw event paths $[\![D_{0\chi_0}^{\rho_0}]\!]_\mathrm{raw} =$

$$\{[\mathtt{lab}_0, \mathtt{lab}_1, \ldots, \mathtt{lab}_n, \ldots] \mid D_{0\chi_0}^{\rho_0}\, \emptyset \xrightarrow{\mathtt{lab}_0} D_{1\chi_1}^{\rho_1}\, \mu_1 \xrightarrow{\mathtt{lab}_1} \ldots \xrightarrow{\mathtt{lab}_{n-1}} D_{n\chi_n}^{\rho_n}\, \mu_n \xrightarrow{\mathtt{lab}_n} \ldots\}$$

progress $[\![D_{0\chi_0}^{\rho_0}]\!]_\mathrm{progress} = \{\pi \mid \pi \in [\![D_{0\chi_0}^{\rho_0}]\!]_\mathrm{raw}$ *and for all configurations* $D_{i\chi_i}^{\rho_i}\mu_i$ *arising in* π, *if* $D_i = R[D]$ *for some* R, D, *then there is a later configuration* $R'[D]_{\chi_{i+j}}^{\rho_{i+j}}\, \mu_{i+j} \xrightarrow{\mathtt{lab}_{i+j}}$ *with redex the same subterm occurrence* $D\}$.

fair $[\![D_{0\chi_0}^{\rho_0}]\!]_\mathrm{fair} = \{\pi \mid \pi \in [\![D_{0\chi_0}^{\rho_0}]\!]_\mathrm{progress}$ *and each event* e *placed in* μ *during* π*'s computation is eventually removed from* μ *at some later point in the computation* $\}$

interaction paths $[\![D_\chi^\rho]\!]_\mathrm{IP} = \{\pi_\chi^\rho \mid \pi_0 \in [\![D_\chi^\rho]\!]_\mathrm{fair}$, π *is* π_0 *with all events not of the form* $\mathtt{in}(e)/\mathtt{out}(e)$ *removed, and the events* $\mathtt{in}(e)/\mathtt{out}(e)$ *in* π *are all unique* $\}$.

The semantics of a top-level diagram, $[\![D_\chi^\rho]\!]$, *is then* $[\![\{D\}_\chi^\rho]\!]_\mathrm{IP}$.

$$D^{\rho}_{\chi}\,\mu \xrightarrow{\texttt{in}(e)} D^{\rho}_{\chi'}\,(\mu \cup \{e\})$$

where $e \notin \mu$, $\chi' = \chi \cup (acq(e) - \rho)$, $pkt(e)$ is not keyed, $target(e) \in \rho$, and $(acq(D) \cup acq(\mu)) \cap acq(e) \subseteq \rho \cup \chi$

$$D^{\rho}_{\chi}\,(\mu \cup \{e\}) \xrightarrow{\texttt{out}(e)} D^{\rho'}_{\chi}\,\mu$$

where $e \notin \mu$, $\rho' = \rho \cup (acq(e)) - \chi)$, $pkt(e)$ is not keyed, and $target(e) \in \chi$

$$R[\texttt{skip}; D]^{\rho}_{\chi}\,\mu \xrightarrow{\texttt{seq}} R[D]^{\rho}_{\chi}\,\mu$$

$$R[\texttt{skip} \parallel \texttt{skip}]^{\rho}_{\chi}\,\mu \xrightarrow{\texttt{par}} R[\texttt{skip}]^{\rho}_{\chi}\,\mu$$

$$R[D_l \oplus D_r]^{\rho}_{\chi}\,\mu \xrightarrow{\texttt{choose}(l)} R[D_l]^{\rho}_{\chi}\,\mu$$

and similarly for $\texttt{choose}(r)$

$$R[\texttt{fork}(D)]^{\rho}_{\chi}\,\mu \xrightarrow{\texttt{fork}} (R[\texttt{skip}] \parallel R'[D])^{\rho}_{\chi}\,\mu$$

where for $R = R_0^-[\{\!|\gamma_1 : \ldots R_n^-[\{\!|\gamma_n : R_{n+1}^-|\!\}] \ldots |\!\}]$, $R' = \{\!|\gamma_1 : \ldots \{\!|\gamma_n : \bullet|\!\} \ldots |\!\}$

$$R[\texttt{receive}(a_{\mathrm{d}} \triangleleft M_{\mathrm{d}}\{ : \kappa\})]^{\rho}_{\chi}\,(\mu \cup \{e\}) \xrightarrow{\texttt{receive}(e)} R'[\texttt{skip}]^{\rho}_{\chi}\,\mu$$

where $R' = R \oplus \overline{x := s}$, and $pkt(e) = a_{\mathrm{d}}(\varepsilon(R)) \triangleleft M_{\mathrm{d}}(\varepsilon(R'))\{ : \kappa\}$

$$R[\texttt{send}(a_{\mathrm{d}} \triangleleft M_{\mathrm{d}}\{ : \kappa\})]^{\rho}_{\chi}\,\mu \xrightarrow{\texttt{send}(e)} R[\texttt{skip}]^{\rho}_{\chi}\,(\mu \cup \{e\})$$

where $pkt(e) = a_{\mathrm{d}}(\varepsilon(R)) \triangleleft M_{\mathrm{d}}(\varepsilon(R))\{ : \kappa\}$ and e is a fresh event

$$R[\texttt{new}(x)]^{\rho}_{\chi}\,\mu \xrightarrow{\texttt{new}(s)} (R \oplus x := s)[\texttt{skip}]^{\rho}_{\chi}\,\mu$$

where $acq(s) \subseteq \rho \cup \chi \cup acq(R) \cup acq(\mu)$

$$R[\texttt{fresh}(x)]^{\rho}_{\chi}\,\mu \xrightarrow{\texttt{fresh}(a)} (R \oplus x := s)[\texttt{skip}]^{\rho}_{\chi}\,\mu$$

where $a \notin \rho \cup \chi \cup acq(R[\texttt{skip}]) \cup acq(\mu)$

$$R[\texttt{constrain}(\phi)]^{\rho}_{\chi}\,\mu \xrightarrow{\texttt{constrain}} R[\texttt{skip}]^{\rho}_{\chi}\,\mu$$

where $\phi(\rho, \chi, \varepsilon(R))$ holds

$$R[x := \psi]^{\rho}_{\chi}\,\mu \xrightarrow{\texttt{assign}} (R'[\texttt{skip}]^{\rho}_{\chi}\,\mu$$

where $R' = R\,@\,x := \psi(\rho, \chi, \varepsilon(R))] \neq [\,]$

$$R[\texttt{rec}\,X.D]^{\rho}_{\chi}\,\mu \xrightarrow{\texttt{recursion}} R[D[(\texttt{rec}\,X.D)/X]]^{\rho}_{\chi}\,\mu$$

$$R[\{D\}]^{\rho}_{\chi}\,\mu \xrightarrow{\texttt{scope-in}} R[\{\!|(\lambda x.\bot) : D|\!\}]^{\rho}_{\chi}\,\mu$$

$$R[\{\!|\gamma : \texttt{skip}|\!\}]^{\rho}_{\chi}\,\mu \xrightarrow{\texttt{scope-out}} R[\texttt{skip}]^{\rho}_{\chi}\,\mu$$

Fig. 2. Single-Step Computation for Diagrams

4.3 Commentary on the Definition

The single-step computation rules themselves are for the most part familiar territory for operational semantics presentations, with a few different aspects. They represent a language with syntax something like CSP, and asynchronous message passing and name handling modelled on the actor approach. We provide some commentry on what are perhaps some of the nonstandard aspects of the rules.

The `par` rule could just as well map `skip` $\|$ D to D. The `receive` rule contains implicit pattern-matching. Diagram variables in M_{d} are considered pattern variables, and are matched against the event e. Note that the receiver a_{d} could itself be a diagram variable, but this is not considered part of the pattern: it is not possible to receive a message destined for an arbitrary actor. $acq(\bar{s}) \subseteq acq(e)$ holds by the pattern match. In both send and receive, the keys κ are not "κ_{d}"–they are simple constants and cannot be variables which are defined in the environment. The `new` rule allocates variable x at the current lexical scoping level, and gives it an arbitrary value based on the names of actors currently known. `fresh`, on the other hand, assigns a single actor name to x which is not currently known. In either rule, if x were already declared within the current lexical level, the old value is replaced. `assign` updates a variable based on ψ. The side condition only fails for the case a variable is assigned to which was never declared; this will cause the computation to get stuck and thus ruled out by lack of progress. `constrain` only continues to compute when the constraint holds; if not the computation is stuck and is ruled out by the progress requirement.

The most unusual aspect of the semantics lies in the details of the progress requirement. This requirement is significantly stronger than standard fairness requirements, and makes the computation system unrealizable. In fact, even without requiring progress the computation system is unrealizable because predicates $\phi(\rho, \chi, \varepsilon)$ may be undecidable, and thus for instance a deterministic halting-problem solver may be defined. However, assuming predicates must be decidable, the progressing paths still may not be realized by some actor computation. An example of such a non-realizable diagram is:

$$
\begin{array}{l}
\texttt{new}(\texttt{nomorezeros} = \mathit{false});\\
[\ (\texttt{receive}(a \triangleleft 0);\ \texttt{constrain}(\neg\texttt{nomorezeros});\\
\ \texttt{nomorezeros} := \mathit{true};\ \texttt{send}(c \triangleleft 1))\\
\oplus\\
(\texttt{receive}(a \triangleleft x);\ \texttt{constrain}(\neg(x = 0 \wedge \texttt{nomorezeros}));\\
\ \texttt{send}(c \triangleleft 0))\ ]^{0..\infty}
\end{array}
$$

it replies 0 to all inputs, except the *last* 0 input *may* get a 1 reply. No real-

izable system can forsee the future to know when the last input of a particular form has arrived.

Progress rules out any computation path which contains a parallel computation that is stuck, *i.e.*, does not reduce. The phrase "subterm occurrence", in analogy to the concept of *residual* in the lambda-calculus, denotes a particular occurrence of a subterm, because the same syntax may in theory occur multiple times in a single term. Each ocurrence must progress. A fully formal definition may be obtained by decorating each subterm with a unique label.

Particular computation paths that progress rules out include

- paths with false constraints such as $R[\texttt{constrain}(x = 0)]^{\rho}_{\chi}\,\mu$ where $R@x = 1$;
- paths which attempt receipt of a message on an unused local actor name, such as $R[\texttt{receive}(a \lhd M_{\mathrm{d}}\{ : \kappa\})]^{\rho}_{\chi}\,\emptyset$ for actor $a \notin \rho \cup \chi$ that is never sent out of the configuration and which is sent no messages locally;
- paths which attempt receipt of a message, $R[\texttt{receive}(a \lhd M_{\mathrm{d}}\{ : \kappa\})]^{\rho}_{\chi}\,\emptyset$ where in this particular computation path the environment will in no such message and it will not be sent locally either;
- paths which attempt receipt of a message on an external actor name, or send of a keyed packet to an external actor;
- assignments to variables that do not exist in the environment, $R[x := 0]^{\rho}_{\chi}\,\mu$ for $R@x = \bot$.

The above cases (excepting the first) are often a product of an ill-conceived diagram design. However, There is no firm line that may be drawn between the well-conceived and ill-conceived diagrams: well-conceived diagrams, when composed with other diagrams, may appear ill-conceived. Nonetheless a simple conservative aproximation of ill-conceived diagrams is possible that detects many obvious errors.

Definition 4.4 *A diagram* D^{ρ}_{χ} *is* ill-conceived *if* $[\![D^{\rho}_{\chi}]\!] = \emptyset$. *Other diagrams besides these may reasonably be classified as ill-conceived.*

If a diagram has no paths, it cannot be sensible. There are other diagrams which are intuitively not sensible, but there is no firmer line that can be drawn between sensible and not. Consider D that contain a subterm occurrence $D_l \oplus D_r$ for which all computation paths in $[\![\{D\}^{\rho}_{\chi}]\!]_{\mathrm{fair}}$ either invariably reduce this occurrence redex by $\texttt{choice}(l)$, or invariably by $\texttt{choice}(r)$. These choice operators may thus be simplified away to the always-chosen case only, and this may be due to an ill-conceived expression in the case never taken. However, it could also be due to the fact that in the context the subterm occurs in, the path not taken is not used, because the specification is more general than the current usage. For this reason, such cases are not invariably classified as ill-conceived.

An example which shows a need for keys is

$$\begin{aligned}&\texttt{new}(x);((x := 1; \texttt{send}(a \triangleleft x : \kappa) \oplus x := 2; \texttt{send}(a \triangleleft x : \kappa')) \parallel \\ &\quad (\texttt{receive}(a \triangleleft x : \kappa) \oplus \\ &\quad (\texttt{receive}(a \triangleleft x : \kappa'); (\texttt{skip} \oplus (\texttt{constrain}(x \neq 2); \texttt{send}(a \triangleleft \texttt{BAD}))))))\end{aligned}$$

—if the keys were removed, a κ-κ' communication could occur and BAD could be sent to a.

5 Toward an Algebra of Diagrams

We now outline the algebra of diagrams; work remains to be done in this area. See [Tal97] for full definition of the algebra of interaction path sets; basic definitions were given previously in section 3. The algebra on diagrams is directly lifted from the algebra on *Ip* sets vie the semantic meaning function for diagrams, $[\![D^{\rho}_{\chi}]\!]$. When performing algebraic reasoning on diagrams, we use the convention that D^{ρ}_{χ} in fact stands for its interaction path semantics, $[\![D^{\rho}_{\chi}]\!]$. The algebraic operations on diagrams are inherited from the algebraic operations on interaction path sets:

Definition 5.1 (Composition, Restriction, Renaming)

$$\begin{aligned}D_1{}^{\rho_1}_{\chi_1} \parallel D_2{}^{\rho_2}_{\chi_2} \quad &\textit{means} \quad [\![D_1{}^{\rho_1}_{\chi_1}]\!] \parallel [\![D_2{}^{\rho_2}_{\chi_2}]\!] \\ D^{\rho}_{\chi}\lceil\rho' \quad &\textit{means} \quad [\![D^{\rho}_{\chi}]\!]\lceil\rho' \\ \widehat{\alpha}(D^{\rho}_{\chi}) \quad &\textit{means} \quad \widehat{\alpha}([\![D^{\rho}_{\chi}]\!])\end{aligned}$$

The notion of equivalence desired for top-level diagrams is an *extensional* one: we are not interested in internal structure of the diagrams, only that an actor configuration satisfies one specification if and only if it satisfies another. Thus two diagrams are defined to be equivalent when their interaction paths are the same.

Definition 5.2 (Extensional Equivalence of Diagrams) $D_1{}^{\rho_1}_{\chi_1} \cong D_2{}^{\rho_2}_{\chi_2}$ *iff* $[\![D_1{}^{\rho_1}_{\chi_1}]\!] = [\![D_2{}^{\rho_2}_{\chi_2}]\!]$.

Note that we use $\cong$ for extensional equivalence, reserving $=$ for syntactic identity of diagrams.

The composition of top-level diagrams is achieved just by forming a new diagram which places the composed diagrams in parallel. This allows for modular construction of diagrams and modular reasoning about diagram properties.

Theorem 5.3

$$D_1{}^{\rho_1}_{\chi_1} \parallel D_2{}^{\rho_2}_{\chi_2} \cong (\{D_1\} \parallel \{D_2\})(^{\rho_1}_{\chi_1} \parallel {}^{\rho_2}_{\chi_2}),$$

provided $\overset{\rho_1}{\chi_1} \bowtie \overset{\rho_2}{\chi_2}$ *and for each* $\texttt{receive}(a_d, mp_d)$ *occurring in* D_1*, by inspection,* $a_d \notin \rho_2$*, and similarly for* D_2*.*

The "by inspection" condition perhaps needs elaborating. One conservative interpretation would be that either $a_d = a$ for $a \in \rho_1$, or $a_d = x$ and $\varepsilon(x) \in \rho_1$ for all ε arising in $[\![D_1{}^{\rho}_{\chi}]\!]$. This condition is needed to guarantee messages are destined for one component or the other, and not both. In section 2 the `FunctionComposer` uses this theorem to show its components may be defined as separate top-level diagrams and composed. After components are composed, `send` and `receive` messages between the two components may be matched to give `send-receive` edges. The function composer example also illustrates this transformation. A future goal is a fully rigorous justification of this operation.

Restriction is elementary if newly restricted receptionists were in fact sent no messages in the specification:

Lemma 5.4

$$D^{\rho\cup\rho'}_{\chi} \lceil \rho \cong D^{\rho}_{\chi}$$

provided for each $\texttt{receive}(a_d, mp_d)$ *occurring in* D*, by inspection,* $a_d \notin \rho'$*.*

What is defined here is equivalence on top-level diagrams. In general it will be desirable to define equivalance on diagram fragments; the natural idea is to use some sort of contextual equivalence *a la* Plotkin. We leave that topic for future work.

6 Related Work

There are a wide variety of notations for concurrent/distributed system specification. Different forms of specification have different strengths and weaknesses, and for large systems a number of different techniques will probably be needed in parallel. We briefly review some of the current schools by way of background.

Process Algebras

Process algebra notation may be used to formally specify the communication actions of concurrent systems, and this was in fact one of the original goals of CCS [Mil80]. Process algebra and specification diagrams in fact share some significant similarities. Parallel composition is of a similar sort in both; choice in specification diagrams could be viewed as a generalization of CCS' external choice operator. message send and receive is analogous to the related concepts in the π-calculus [MPW92,HT91], although the π-calculus restricts data passed to be a channel name, and is in the classical presentation, synchronous as opposed to asynchronous. Name-passing and dynamic name creation are important to distributed systems and are treated in specification diagrams

as well as the π calculus. The trace-based semantic framework is a concept shared with CSP [Hoa85].

There are differences as well, and the most important ones are found beneath the surface in the semantics of operators and not their syntax. The object-based behavior of specification diagrams is enforced by the interfaces; for this there is no analogue in process algebra since it is not object-based. There is a subtle difference in the meaning given to specification diagrams in comparison to process algebra. Simply put, process algebra is given a purely operational, realizable, interpretation. Even though specification diagrams have an operational semantics, this semantics is not realizable. If a constraint fails during computation, the computation *never happened*; it disappears from the set of possible paths. Constraints themselves may not be decidable properties. Specifications written in process algebra notation admit the possibility of deadlock since the environment may not send a particular desired message. Specification diagrams, on the other hand, constrain the environment so that deadlock implicitly cannot occur; instead, either a specification is ill-formed, or composition of specifications will fail. Deadlock can in fact be *specified* in specification diagrams, by actively ignoring all input. The `Sink` example earlier is such an example. Specification diagrams allow communication to be constrained both at send and receive by cross-edges, so operationally speaking, if a message is not received by its intended receiver, that computation path never happened. There are advantages and disadvantages of uncomputable specification languages. The main advantage of uncomputable languages is their expressivity. The main advantage of computable languages is they generally possess more decidable properties.

Choice in specification diagrams could be called "extremely external" if the nomenclature of internal/external choice of CCS is used. Internal choice is a random coin flip which irrevocably picks one of two paths. External choice in its general form is a guarded choice; the path chosen must have the guard condition holding. In specification diagrams, the constraints allow a choice to be "un-chosen" even after it had been started, not just at the beginning. This is not an operational notion, but is useful in certain cases to allow for succinct specification of concurrent object behavior.

$$\texttt{in}(a_{\mathrm{d}} \triangleleft x);\ (\texttt{out}(a'_{\mathrm{d}} \triangleleft x);\ \texttt{in}(a_{\mathrm{d}} \triangleleft y);\ \texttt{constrain}(y > 0)) \oplus \texttt{skip}$$

–This specification has odd behavior of only forwarding a message when the *next* message is a positive number.

A number of full specification languages based on process algebra have been developed; examples include LOTOS [BB87], which is based on CSP; it is now an an ISO standard. Esterel [BG92] is a process algebra based specification language with a synchronous execution semantics.

Temporal Logic

Temporal logic formulae have been extensively used as a means for logical specification of concurrent and distributed systems [JM86,MP92,Lam94]. While logics may express an extremely broad collection of properties, a significant disadvantage is the need for large, complex formulae to specify nontrivial systems: readability of specifications becomes a serious issue even for small specifications, and users thus require more advanced training. Specification diagrams are not a logic; as such, they cannot logically assert global properties of programs, only local properties via constraints. The equational theory of specification diagrams will provide the basis for more abstract reasoning about actor system components.

Automata-Based Formalisms

Finite automata are useful for specifying systems which have a strong state-based behavior. They lack expressivity, but make up for this lack by their amenability to automatic verification by state-space search techniques.

The StateCharts formalism [Har87] has become particularly popular in industry. States of the automaton represent states of the system (where certain invariant properties hold), and state transitions represent actions. The StateCharts formalism has features beyond simple finite automata, including the ability to nest and compose automata. This syntactic sugar makes it feasible to write specifications. Automata are also graphical and so serve as good visual specifications. Their primary weakness is that a complex software system may not have a meaningful global state, and properties of such systems are more naturally expressed in terms of events and relations on events. UML notation includes a StateCharts-based style of diagram. A formal semantics of StateCharts has been defined [HPPSS87], but the tools are not sound with respect to a formal semantics and so the effort is not completely satisfactory.

Message-Passing Diagrams

Message-passing diagrams are a common form of informal graphical specification. A message-passing diagram has a time-line showing the message-passing behavior between different components. Unlike the other approaches described above, message passing specifications are usually object-based and can be asynchronous. The UML Sequence Diagram [RJB98] (derived in turn from the event trace diagram of [*et al*91]) is a simple form of message passing diagram for rpc-style communication. Specfication diagrams can be viewed as a major extension of UML sequence diagram notation. In the actor model, event diagrams [Gre75,Hew77] model actor computation in terms of message-passing between actors. Clinger[Cli81] formalizes event diagrams as mathematical structures and defines a formal semantics mapping actor system descriptions to sets event diagrams. More generally sets of event diagrams can be thought of as abstract specifications. These have rich mathematical structure but, are in general highly undecidable. Specification diagrams were

partly inspired by event diagrams, and can be viewed as a condensation of a possibly infinite set of possibly infinite-sized event diagrams to one, finite, representation.

Acknowledgements

Thanks to Carolyn Talcott for many discussions and for comments on several versions of this document. Also thanks to Gul Agha and Ian Mason for helpful comments.

References

[Agh86] G. Agha. *Actors: A Model of Concurrent Computation in Distributed Systems.* MIT Press, Cambridge, Mass., 1986.

[AMST97] G. Agha, I. A. Mason, S. F. Smith, and C. L. Talcott. A foundation for actor computation. *Journal of Functional Programming*, 7:1–72, 1997.

[BB87] T. Bolognesi and E. Brinksma. Introduction to the ISO specification language LOTOS. *Computer Networks and ISDN Systems*, 14:25–59, 1987.

[BG92] G. Berry and G. Gonthier. The ESTEREL synchronous programming language: design, semantics, implementation. *Science of Computer Programming*, 19(2):87–152, November 1992.

[Cli81] W. D. Clinger. *Foundations of Actor Semantics.* PhD thesis, MIT, 1981. MIT Artificial Intelligence Laboratory AI-TR-633.

[*et al*91] J. Rumbaugh *et al.* *Object-Oriented Modeling and Design.* Prentice-Hall, 1991.

[Gre75] I. Greif. Semantics of communicating parallel processes. Technical Report 154, MIT, Project MAC, 1975.

[Har87] D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, June 1987.

[Hew77] C. Hewitt. Viewing control structures as patterns of passing messages. *Journal of Artificial Intelligence*, 8(3):323–364, 1977.

[Hoa85] C. A. R. Hoare. *Communicating Sequential Processes.* Prentice-Hall, 1985.

[HPPSS87] D. Harel, A. Pnueli, J. Pruzan-Schmidt, and R. Sherman. On the formal semantics of statecharts. In *Proceedings* 2^{nd} *Annual Symposium on Logic in Computer Science,* Ithaca, New York, pages 54–64, 1987.

[HT91] Kohei Honda and Mario Tokoro. An object calculus for asynchronous communication. In Pierre America, editor, *Proceedings of the European*

Conference on Object-Oriented Programming (ECOOP), volume 512 of *Lecture Notes in Computer Science*, pages 133–147. Springer-Verlag, Berlin, Heidelberg, New York, Tokyo, 1991.

[JM86] Farnam Jahanian and Aloysius Mok. Safety analysis of timing properties in real-time systems. *IEEE Transaction on Software Engineering*, 12(9):890–904, 1986.

[Lam94] L. Lamport. The temporal logic of actions. *ACM TOPLAS*, 16(3):872–923, May 1994.

[Mil80] Robin Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer Verlag, 1980.

[MP92] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems: Specification*. Springer Verlag, 1992.

[MPW92] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes (Parts I and II). *Information and Computation*, 100:1–77, 1992.

[MT97] I. A. Mason and C. L. Talcott. A semantically sound actor translation, 1997. submitted.

[RJB98] Jim Rumbaugh, Ivar Jacobson, and Grady Booch. *Unified Modeling Language Reference Manual*. Addison-Wesley, 1998.

[Tal97] C. L. Talcott. Composable semantic models for actor theories. In T. Ito M. Abadi, editor, *Theoretical Aspects of Computer Science*, number 1281 in Lecture Notes in Computer Science, pages 321–364. Springer-Verlag, 1997.

Electronic Notes in Theoretical Computer Science 10 (1998)
URL: http://www.elsevier.nl/locate/entcs/volume10.html 4 pages

Mobile Ambients

(Extended Abstract)

Luca Cardelli [1]

Digital Equipment Corporation
Systems Research Center

Andrew D. Gordon [1]

University of Cambridge
Computer Laboratory

There are two distinct areas of work in mobility: mobile computing, concerning computation that is carried out in mobile devices (laptops, personal digital assistants, etc.), and mobile computation, concerning mobile code that moves between devices (applets, agents, etc.). We aim to describe all these aspects of mobility within a single framework that encompasses mobile agents, the ambients where agents interact and the mobility of the ambients themselves.

This extended abstract of a longer paper [2] presents a minimal calculus of ambients that includes only mobility primitives. Section 1 gives its syntax and introduces its semantics informally. Section 2 gives a formal semantics. Section 3 concludes.

1 Syntax and Informal Semantics

We begin by defining the syntax of the calculus in the following table. The main syntactic categories are processes (including ambients and agents that execute actions) and capabilities.

Mobility and Communication Primitives

n	names
$M ::=$	capability
$\quad in\ n$	$\quad$ can enter into n
$\quad out\ n$	$\quad$ can exit out of n
$\quad open\ n$	$\quad$ can open n
$P, Q, R ::=$	process
$\quad (\nu n)P$	$\quad$ restriction

[1] Current affiliation: Microsoft Research.

$\mathbf{0}$	inactivity
$P \mid Q$	composition
$!P$	replication
$n[P]$	ambient
$M.P$	action

The first four process primitives (restriction, inactivity, composition and replication) have the same meaning as in the π-calculus [4] namely: restriction is used to introduce new names and limit their scope; $\mathbf{0}$ has no behavior; $P \mid Q$ is the parallel composition of P and Q; and $!P$ is an unbounded number of parallel replicas of P. The main difference with respect to the π-calculus is that names are used to name ambients instead of channels. To these standard primitives we add ambients, $n[P]$, and the exercise of capabilities, $M.P$.

We now introduce the semantics of ambients informally. A reduction relation $P \rightarrow Q$ describes the evolution of a process P into a new process Q.

Ambients

An ambient is written $n[P]$, where n is the name of the ambient, and P is the process running inside the ambient. In $n[P]$, it is understood that P is actively running, and that P can be the parallel composition of several processes. We emphasize that P is running even when the surrounding ambient is moving. Running while moving may or may not be realistic, depending on the nature of the ambient and of the communication medium through which the ambient moves, but it is consistent to think in those terms.

In general, an ambient exhibits a tree structure induced by the nesting of ambient brackets. Each node of this tree structure may contain a collection of (non-ambient) processes running in parallel, in addition to subambients. We say that these processes are running in the ambient, in contrast to the ones running in subambients.

Nothing prevents the existence of two or more ambients with the same name, either nested or at the same level. Once a name is created, it can be used to name multiple ambients. Moreover, $!n[P]$ generates multiple ambients with the same name. This way, for example, one can easily model the replication of services.

Operations that change the hierarchical structure of ambients are sensitive. In particular such operations can be interpreted as the crossing of firewalls or the decoding of ciphertexts. Hence these operations are restricted by capabilities. Thanks to capabilities, an ambient can allow other ambients to perform certain operations without having to reveal its true name.

Actions and Capabilities

The process $M.P$ executes an action regulated by the capability M, and then continues as the process P. The process P does not start running until the

action is executed. The reduction rules for $M.P$ depend on the capability M, and are described below case by case.

We consider three kinds of capabilities: one for entering an ambient, one for exiting an ambient and one for opening up an ambient. Capabilities are obtained from names; given a name n, the capability $in\ n$ allows entry into n, the capability $out\ n$ allows exit out of n and the capability $open\ n$ allows the opening of n. Implicitly, the possession of one or all of these capabilities for n is insufficient to reconstruct the original name n.

An entry capability, $in m$, can be used in the action $in m.P$, which instructs the ambient surrounding $in\ m.P$ to enter a sibling ambient named m. If no sibling m can be found, the operation blocks until a time when such a sibling exists. If more than one m sibling exists, any one of them can be chosen. The reduction rule is:

$$n[in\ m.P \mid Q] \mid m[R] \rightarrow m[n[P \mid Q] \mid R]$$

If successful, this reduction transforms a sibling n of an ambient m into a child of m. After the execution, the process $in\ m.P$ continues with P, and both P and Q find themselves at a lower level in the tree of ambients.

An exit capability, $out\ m$, can be used in the action $out\ m.P$, which instructs the ambient surrounding $out\ m.P$ to exit its parent ambient named m. If the parent is not named m, the operation blocks until a time when such a parent exists. The reduction rule is:

$$m[n[out\ m.P \mid Q] \mid R] \rightarrow n[P \mid Q] \mid m[R]$$

If successful, this reduction transforms a child n of an ambient m into a sibling of m. After the execution, the process $in\ m.P$ continues with P, and both P and Q find themselves at a higher level in the tree of ambients.

An opening capability, $open\ n$, can be used in the action $open\ n.P$. This action provides a way of dissolving the boundary of an ambient named n located at the same level as $open\ n.P$, according to the rule:

$$open\ n.P \mid n[Q] \rightarrow P \mid Q$$

If no ambient n can be found, the operation blocks until a time when such an ambient exists. If more than one ambient n exists, any one of them can be chosen.

2 Formal Semantics

We now give an operational semantics of our calculus, based on a structural congruence between processes, $\equiv$, and a reduction relation $\rightarrow$. This is a semantics in the style of Milner's reaction relation [3] for the π-calculus, which was itself inspired by the Chemical Abstract Machine of Berry and Boudol [1].

We let *structural congruence*, $\equiv$, be the least relation on processes that satisfies the following equations and rules:

Structural Congruence

$P \equiv P$	$P \mid Q \equiv Q \mid P$
$Q \equiv P \Rightarrow P \equiv Q$	$(P \mid Q) \mid R \equiv P \mid (Q \mid R)$
$P \equiv Q, Q \equiv R \Rightarrow P \equiv R$	$!P \equiv P \mid !P$
	$(\nu n)(\nu m)P \equiv (\nu m)(\nu n)P$
$P \equiv Q \Rightarrow (\nu n)P \equiv (\nu n)Q$	$(\nu n)(P \mid Q) \equiv P \mid (\nu n)Q$ if $n \notin fn(P)$
$P \equiv Q \Rightarrow P \mid R \equiv Q \mid R$	$(\nu n)(m[P]) \equiv m[(\nu n)P]$ if $n \neq m$
$P \equiv Q \Rightarrow !P \equiv !Q$	$P \mid \mathbf{0} \equiv P$
$P \equiv Q \Rightarrow n[P] \equiv n[Q]$	$(\nu n)\mathbf{0} \equiv \mathbf{0}$
$P \equiv Q \Rightarrow M.P \equiv M.Q$	$!\mathbf{0} \equiv \mathbf{0}$

We let the *reduction relation*, $\rightarrow$, be the least relation on processes to satisfy the following rules:

Reduction

$n[in\ m.P \mid Q] \mid m[R] \rightarrow m[n[P \mid Q] \mid R]$	$P \rightarrow Q \Rightarrow P \mid R \rightarrow Q \mid R$
$m[n[out\ m.P \mid Q] \mid R] \rightarrow n[P \mid Q] \mid m[R]$	$P \rightarrow Q \Rightarrow (\nu n)P \rightarrow (\nu n)Q$
$open\ n.P \mid n[Q] \rightarrow P \mid Q$	$P \rightarrow Q \Rightarrow n[P] \rightarrow n[Q]$
$P' \equiv P, P \rightarrow Q, Q \equiv Q' \Rightarrow P' \rightarrow Q'$	

3 Summary

We presented the syntax and semantics of a minimal calculus of ambients. In the full paper [2] we provide more motivation and intuition, investigate a calculus with additional primitives for I/O, survey related work and include many examples. We discuss how the notion of a mobile ambient captures the structure of complex networks and the behavior of mobile computation. The full calculus is no more complex than common process calculi, but supports reasoning about mobility and, at least to some degree, security.

References

[1] G. Berry and G. Boudol. The chemical abstract machine. *Theoretical Computer Science*, 96(1):217–248, April 1992.

[2] A. D. Gordon and L. Cardelli. Mobile ambients. In *Foundations of System Specification and Computation Structures*, Lecture Notes in Computer Science. Springer-Verlag, 1998. To appear.

[3] R. Milner. Functions as processes. *Mathematical Structures in Computer Science*, 2:119–141, 1992.

[4] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, parts I and II. *Information and Computation*, 100:1–40 and 41–77, 1992.

Electronic Notes in Theoretical Computer Science 10 (1998)
URL: http://www.elsevier.nl/locate/entcs/volume10.html 4 pages

A Type System for Object Initialization In the Java Bytecode Language (summary)

Stephen N. Freund John C. Mitchell

Department of Computer Science
Stanford University
Stanford, CA 94305-9045
{freunds, mitchell}@cs.stanford.edu

Abstract

In the standard Java implementation, a Java language program is compiled to Java bytecode and this bytecode is then interpreted by the Java Virtual Machine. Since bytecode may be written by hand, or corrupted during network transmission, the Java Virtual Machine contains a *bytecode verifier* that performs a number of consistency checks before code is interpreted. As one-step towards a formal specification of the verifier, we describe a precise specification of a subset of the bytecode language dealing with object creation and initialization.

1 Introduction

The Java programming language is a statically-typed general-purpose programming language with an implementation architecture that is designed to facilitate transmission of compiled code across a network. In the standard implementation, a Java language program is compiled to Java bytecode and this bytecode is then interpreted by the Java Virtual Machine. The intermediate bytecode language, which we refer to as JVML, is a typed, machine-independent form of assembly language with some low-level instructions that reflect specific high-level Java source language constructs. For example, classes are a basic notion in JVML. Since bytecode may be written by hand, or corrupted during network transmission, the Java Virtual Machine contains a *bytecode verifier* that performs a number of consistency checks before code is interpreted. This protects the receiver from certain security risks and various forms of attack.

In this summary, we describe a specification for a fragment of the bytecode language that includes object creation (allocation of memory) and initialization in terms of a type system. This work is based on a prior study of the

bytecodes for local subroutine call and return [2].

2 Object Initialization

As in many other object-oriented languages, the Java implementation creates new objects in two steps. The first step is to allocate space for the object. This usually requires some environment specific operation to obtain an appropriate region of memory. In the second step, user-defined code is executed to initialize the object. In Java, the user initialization code is provided by a constructor defined in the class of the object. Only after both of these steps are completed can a method be invoked on an object.

In the Java source language, allocation and initialization are combined into a single statement. This is illustrated in the following code fragment:

```
Point p = new Point(3);
p.Print();
```

Since every Java object is created by a statement like the one in the first line here, it does not seem difficult to prevent Java source language programs from invoking methods on objects that have not been initialized.

It is much more difficult to recognize initialization-before-use in bytecode. This can be seen by looking at the five lines of bytecode that are produced by compiling the two lines of source code above:

```
1:  new #1 <Class Point>
2:  dup
3:  iconst_3
4:  invokespecial #4 <Method Point(int)>
5:  invokevirtual #5 <Method void Print()>
```

The most striking difference is that memory allocation (line 1) is separated from the constructor invocation (line 4) by two lines of code. The first intervening line, `dup`, duplicates the pointer to the uninitialized object. This line is needed due to the calling convention of the Java Virtual Machine. The second line, `iconst_3`, pushes the constructor argument 3 onto the stack. If the constructor arguments for more complicated, many more lines of code would appear between lines (1) and (4).

Since pointers may be duplicated, as above, and there may be more than one uninitialized object present at any time, some form of aliasing analysis must be used. Sun's Java Virtual Machine Specification [1] describes the alias analysis used by the Sun's JDK verifier. For each line of the bytecode program, some status information is recorded for every local variable and stack location. When a location points to an object that is not known to be initialized in all executions reaching this statement, the status will include not only the property *uninitialized,* but also the line number on which the uninitialized object would have been created. As references are duplicated on the stack and stored and loaded in the local variables, the analysis also duplicates these

$$(new)\quad \frac{\begin{array}{c} P[i] = \texttt{new}\ \sigma \\ F_{i+1} = F_i \\ S_{i+1} = \hat{\sigma}_i \cdot S_i \\ \hat{\sigma}_i \notin S_i \\ \forall y \in Dom(F_i).\ F_i[y] \neq \hat{\sigma}_i \\ i+1 \in Dom(P) \end{array}}{F, S, i \vdash P} \qquad (init)\quad \frac{\begin{array}{c} P[i] = \texttt{init}\ \sigma \\ F_{i+1} = [\sigma/\hat{\sigma}_j]F_i \\ S_i = \hat{\sigma}_j \cdot \alpha \\ S_{i+1} = [\sigma/\hat{\sigma}_j]\alpha \\ j \in Dom(P) \\ i+1 \in Dom(P) \end{array}}{F, S, i \vdash P}$$

$$(use)\quad \frac{\begin{array}{c} P[i] = \texttt{use}\ \sigma \\ F_{i+1} = F_i \\ S_i = \sigma \cdot S_{i+1} \\ i+1 \in Dom(P) \end{array}}{F, S, i \vdash P}$$

Fig. 1. Static semantics.

line numbers. All references having the same line number are assumed to refer to the same object. When an object is initialized, all pointers that refer to objects created at the same line number are set to *initialized.* In other words, all references to objects of a certain type are partitioned into equivalence classes according to what is statically known about each reference.

Since our approach is type based, the status information associated with each reference for the alias analysis is recorded as part of its type.

3 JVML$_i$

Our study uses JVML$_i$, an idealized subset of JVML encompassing basic constructs and object initialization. This section introduces JVML$_i$ and describes the typing rules performing the alias analysis described in the previous section. In this summary, we present only those instructions related to object initialization:

`new` σ: allocates a new, uninitialized object of type σ.

`init` σ: initializes a previously uninitialized object of type σ. This represents calling the constructor of an object.

`use` σ: performs an operation on an initialized object of type σ. This corresponds to several operations in JVML, including method invocation (`invokevirtual`), accessing an instance field (`putfield/getfield`), etc.

3.1 Typing Rules

A program is represented by an array of instructions. A program P is well typed if there exist F and S such that

$$F, S \vdash P,$$

where F is a vector of functions mapping local variables to types such that $F_i[y]$ is the type of local variable y at line i of a program. Likewise, S is a vector of stack types such that S_i is the type of the operand stack at location

i of the program. The judgment which allows us to conclude that a program P is well typed by F and S is

$$(wt\ prog) \qquad \frac{\begin{array}{c} F_1 = F_{\text{TOP}} \\ S_1 = \epsilon \\ \forall i \in Dom(P).\ F, S, i \vdash P \end{array}}{F, S \vdash P}$$

where F_{TOP} is a function mapping all variables to TOP, the type of unusable values. The first two lines of (*wt prog*) constrain the initial conditions for the program's execution. The third line requires that each instruction in the program is well typed according to local judgments for each instruction. The rules for those instructions dealing with object initialization are presented in Figure 1, where σ is some object type and $\hat{\sigma}_i$ is the type given to an uninitialized object of type σ allocated on line i of P. A soundness result is stated and discussed in the full version of this paper.

4 Discussion

Given the need to guarantee type safety for mobile Java code, developing correct type checking and analysis techniques for JVML is crucial. We have built on the previous work of Stata and Abadi to develop such a specification by formulating a sound type system for object initialization. Although our model is still rather abstract, it has already proved effective as a foundation for examining both JVML and existing bytecode verifiers. In fact, a previously unpublished bug in Sun's verifier implementation was found as a result of the analysis performed while studying the soundness proofs for JVML$_i$ extended with subroutines.

Acknowledgement

Thanks to Martín Abadi and Raymie Stata (DEC SRC) for their assistance on this project. Also, we thank Frank Yellin and Sheng Liang for several interesting discussions.

References

[1] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification.* Addison-Wesley, 1996.

[2] Raymie Stata and Martín Abadi. A type system for Java bytecode subroutines. In *Proc. 25th ACM Symposium on Principles of Programming Languages*, January 1998.

Electronic Notes in Theoretical Computer Science 10 (1998)
URL: http://www.elsevier.nl/locate/entcs/volume10.html 2 pages

Secure Implementation of Channel Abstractions (Abstract)

Martín Abadi

ma@pa.dec.com
Digital Equipment Corporation
Systems Research Center

Cédric Fournet

Cedric.Fournet@inria.fr
INRIA Rocquencourt

Georges Gonthier

Georges.Gonthier@inria.fr
INRIA Rocquencourt

Communication in distributed systems often relies on useful abstractions such as channels, remote procedure calls, and remote method invocations. The implementations of these abstractions sometimes provide security properties, in particular through encryption [7,6,9,8,5,12,13]. In this work we study those security properties, focusing on channel abstractions. We introduce a simple high-level language that includes constructs for creating and using secure channels. The language is a variant of the join-calculus [4] and belongs to the same family as the pi-calculus [11,10]. We show how to translate the high-level language into a lower-level language that includes cryptographic primitives (cf. [1–3]). In this translation, we map communication on secure channels to encrypted communication on public channels. We obtain a correctness theorem for our translation; this theorem implies that one can reason about programs in the high-level language without mentioning the subtle cryptographic protocols used in their lower-level implementation.

The full paper will appear in LICS'98.

References

[1] M. Abadi and A. D. Gordon. A calculus for cryptographic protocols: The spi calculus. Technical Report 414, University of Cambridge Computer Laboratory, January 1997. Extended version of both [2] and [3]. A revised version

appeared as Digital Equipment Corporation Systems Research Center report No. 149, January 1998, and an abridged version will appear in *Information and Computation.*

[2] Martín Abadi and Andrew D. Gordon. A calculus for cryptographic protocols: The spi calculus. In *Proceedings of the Fourth ACM Conference on Computer and Communications Security*, pages 36–47, April 1997.

[3] Martín Abadi and Andrew D. Gordon. Reasoning about cryptographic protocols in the spi calculus. In Antoni Mazurkiewicz and Jòzef Winkowski, editors, *Proceedings of the 8th International Conference on Concurrency Theory*, volume 1243 of *Lecture Notes in Computer Science*, pages 59–73. Springer-Verlag, July 1997.

[4] Cédric Fournet and Georges Gonthier. The reflexive chemical abstract machine and the join-calculus. In *Proceedings of POPL '96*, pages 372–385. ACM, January 1996.

[5] Butler Lampson, Martín Abadi, Michael Burrows, and Edward Wobber. Authentication in distributed systems: Theory and practice. *ACM Transactions on Computer Systems*, 10(4):265–310, November 1992.

[6] John Linn. Generic interface to security services. *Computer Communications*, 17(7):476–482, July 1994.

[7] Jonn Linn. RFC 1508: Generic security service application program interface. Web page at `ftp://ds.internic.net/rfc/rfc1508.txt`, September 1993.

[8] Daniel L. McDonald, Bao G. Phan, and Randall J. Atkinson. A socket-based key management API (and surrounding infrastructure). In *Proceedings of INET96*, 1996. On the Web at `http://www.isoc.org/isoc/whatis/conferences/inet/96/`.

[9] Microsoft. CryptoAPI. Web pages at `http://www.microsoft.com/security/tech/`.

[10] Robin Milner. Functions as processes. *Mathematical Structures in Computer Science*, 2:119–141, 1992.

[11] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, parts I and II. *Information and Computation*, 100:1–40 and 41–77, September 1992.

[12] Leendert van Doorn, Martín Abadi, Mike Burrows, and Edward Wobber. Secure network objects. In *Proceedings 1996 IEEE Symposium on Security and Privacy*, pages 211–221, May 1996.

[13] Ann Wollrath, Roger Riggs, and Jim Waldo. A distributed object model for the Java system. *Computing Systems*, 9(4):265–290, Fall 1996.

Program Units as Higher-Order Modules

Matthew Flatt Matthias Felleisen

Rice University

Abstract

We have designed a new module language called *program units*. Units support separate compilation, independent module reuse, cyclic dependencies, hierarchical structuring, and dynamic linking. In this paper, we present untyped and typed models of units.

1 Program Fragments and Units

Programmers consume code fragments to create programs, and produce code fragments for other programs. When managing fragments becomes mechanistic, programmers write programs that assemble and execute these fragments. Some of these programs launch fragments as separate processes. Other programs link several fragments together to produce a new program fragment. And some programs dynamically link fragments into an already-running program. Especially in this last case, the distinction between the program and the fragments that is manages begins to blur. Unfortunately, current programming languages cannot clearly express the interaction between these levels.

Programming with fragments requires a two-phase view of execution: linking followed by evaluation. This separation of phases is important because it enables the separate compilation, analysis, and optimization of fragments. It also suggests separate programming languages: a core language for implementing fragments and a module language for linking them. Much of the recent work on modularity (in particular, work on ML modules [2,10,11,15,16,18,23]) has taken this two-language view and focused on making the linking language flexible.

For MzScheme [7], we designed and implemented an extension of Scheme that carefully *combines* the core programming language with the linking language. In this language, code fragments called *units* are first-class values. The

[1] This research was partially supported by a NSF Graduate Research Fellowship, NSF grants CCR-9619756, CDA-9713032, and CCR-9708957, and a Texas ATP grant.

only primitive operations on units are linking and invocation, which preserves the phase separation for an individual unit, but programmers can exploit the full flexibility of the core language for the application of these operations.

In this paper we present a typed language of program units that supports

- units that import and export type definitions as well as value definitions;
- compound units that link several units together and hide selected details of the constituent units;
- procedure and type definitions with mutual references across unit boundaries;
- dynamic linking of units into a running program;
- separate compilation of units; and
- flexible linking that allows multiple instances of a unit in a single program.

Units accomodate a variety of core languags, such as ML, Ada, Modula-3, Java, Scheme, or C. Each of these languages can benefit by incorporating the unit language: ML's modules disallow mutually-recursive type or function definitions; linking in Ada, Modula-3, and Java is inflexible because linking is specified within a package and relies on a global package namespace;[2] Scheme has no standard module system; and C, like most languages, lacks a standard mechanism for dynamic linking.

Section 2 provides an overview of programming with units, and Section 3 defines the precise type checking and semantics of units. Section 4 briefly considers compilation issues. The last two sections relate our unit language to existing module languages, and put this work into perspective.

2 Programming with Units

The following sub-sections illustrate the basic design elements of our unit language using an informal, semi-graphical language. The examples assume a core language with lexical bloacks and a sub-language of types. The syntax used for the core language mimics that of ML.

2.1 Defining Units

Figure 1 defines a unit called *Database*. In the graphical notation, each unit is drawn as a box with three sections:

- The top section lists the unit's imported types and values. The *Database*

[2] With class loaders, the meaning of the global namespace can be adjusted, but this adjustment must be described indirectly via a loader object rather than directly in the language. Even then, using hardwired names for imported packages prevents linking a single package in multiple contexts.

unit imports the type *info* (with the kind Ω) for data stored in the database, and the function *error* (with the type str→void), for error-handling.

- The middle section contains the unit's type and value definitions and an initialization expression. The latter performs startup actions for the unit at run-time. The *Database* unit defines the type *db* and the functions *new*, *insert*, and *delete* (plus some other definitions that are not shown). Database entries are keyed by strings, so *Database* initializes a hash table for strings with the expression *strTable* := *makeStringHashTable*().
- The bottom section enumerates the unit's exported types and values. The *Database* unit exports the type *db* and the functions *new*, *insert*, and *delete*.

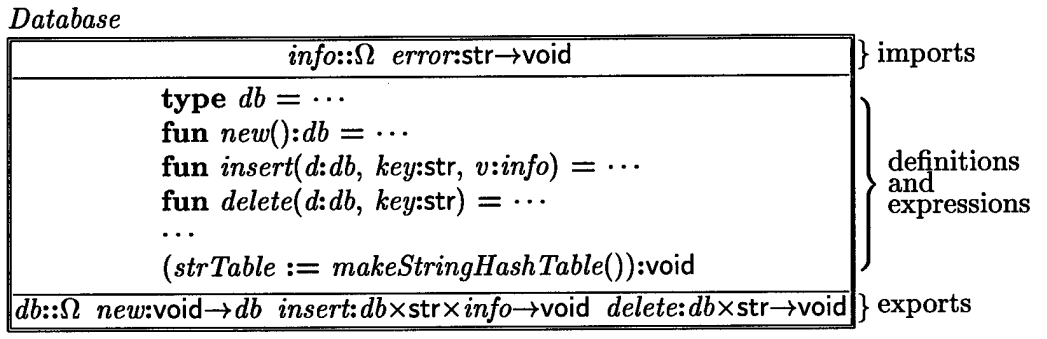


Fig. 1. A simple database unit

In a statically-typed language, all imported and exported values have a type, and all imported and exported types have a kind. Imported and defined types can be used in the the type expressions for imported and exported values. All exported variables must be defined within the unit, and the type expression for an export must use only imported and exported types. In *Database*, both the imported type *info* and the exported type *db* are used in the type expression for *insert*: *db*×str×*info*→void.

A unit is specifically *not* a record of values (as ML structures are usually described). A unit encapsulates unevaluated code, much like the ".o" file created by compiling a C module. Before a unit's definitions and initialization expression can be evaluated, it must be first linked with other units to resolve all of its imports.

2.2 Linking Units

In the graphical notation, units are linked together via arrows connecting the exports of one box with the imports of another. Linking units together creates a compound unit, as illustrated in Figure 2 with the *PhoneBook* unit. This unit links *Database* with *NumberInfo*, a unit that implements the *info* type for phone numbers. The *error* function is not yet determined, so the *PhoneBook*

unit imports *error* and passes the imported value on to *Database*. All of the exported types and values of *Database* and *NumberInfo* are re-exported by *PhoneBook*, except the *delete* function from *Database*. Since the *delete* function is not exported, it is hidden to clients of *PhoneBook*.

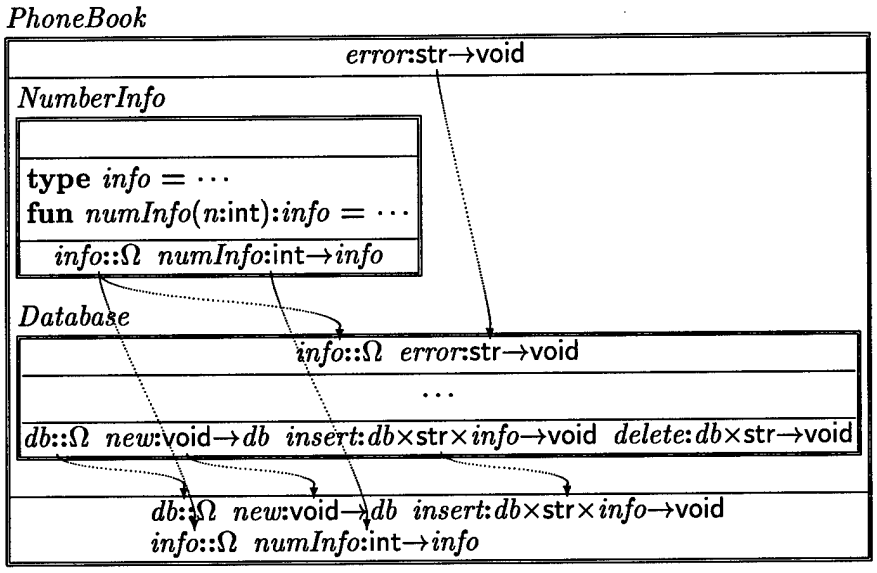


Fig. 2. Linking units to form a compound unit

A complete program is a unit (either simple or compound) without imports. Figure 3 defines a complete interactive phone book program, *InteractivePhoneBook*, which links *PhoneBook* with a graphical interface implementation *Gui* and an error unit *ErrorHandler*. The *Main*[3] contains an initialization expression that creates a database and an associated grahical interface.

A complete program is analogous to an executable file in Unix; *invoking* the unit evaluates the definitions in all of the program's units and then executes their intialization expressions in sequence. Thus, when *InteractivePhoneBook* is invoked, a new phone book database is created and a phone book window is opened by *Main*. The return value of the whole program is the value of the last initialization expression, which is a bool value in *InteractivePhoneBook* (assuming *Main*'s expression is evaluated last).[4]

Since linking and invocation are separate phases, linking can connect mutually recursive functions across units. Figure 4 defines a slightly revised version of the phone book program, *IPB*, where *error* is part of the *Gui* unit. Links flow both from *PhoneBook* to *Gui* and from *Gui* to *PhoneBook*. Thus,

[3] *Main* is not a special name.

[4] Our informal graphical notation does not specify the order of units in a compound unit, but a textual notation covers this aspect of the language.

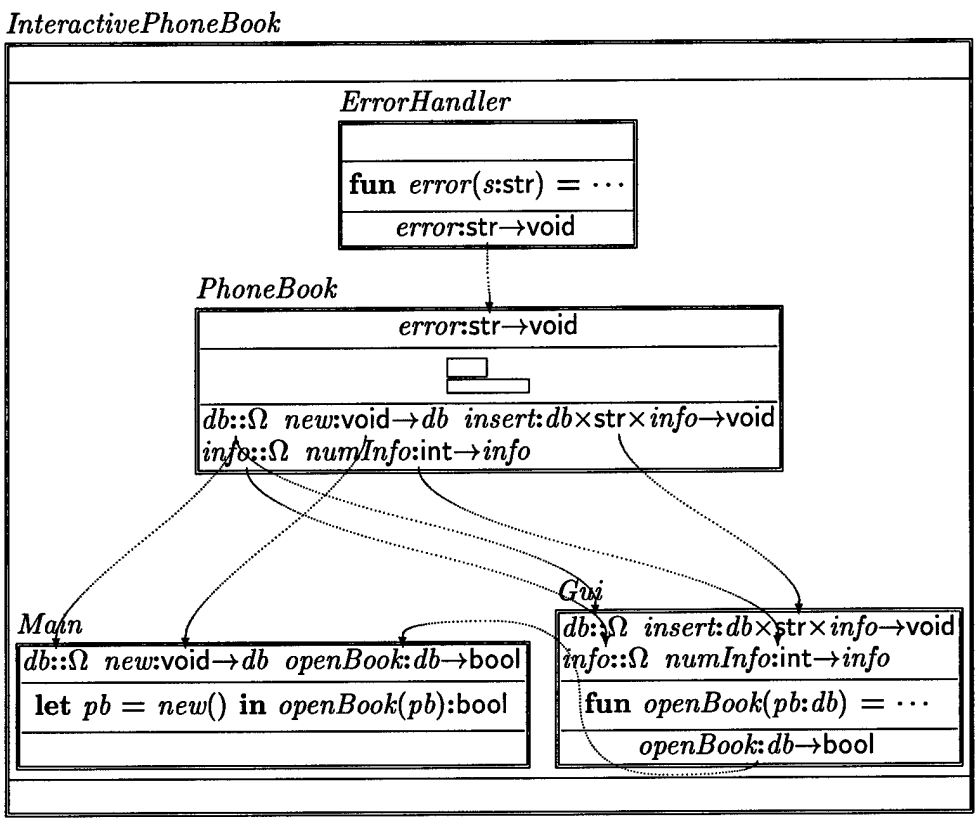


Fig. 3. Linking units to define a complete program

the *insert* function in *PhoneBook* may call *error* in *Gui*, which could in turn call *PhoneBook*'s *insert* again to handle the error.

A compound unit's links must satisfy the type requirements of the constituent units. For example, in *InteractivePhoneBook*, *Main* imports the type *db* from *PhoneBook* unit and also the function *openBook*:*db*→bool from *Gui*. The two occurrences of *db* must refer to the same type. A type checker can verify this by proving that the two occurrences have the same source, which is the *db* exported by *PhoneBook* In contrast, Figure 5 defines a "program" *Bad* in which inconsistent imports are provided to *Main*. Specifically, *db* and *openBook*:*db*→bool refer to types named *db* that originate from different units. The type checker will correctly reject *bad* due to this mismatch.

2.3 Programs that Link and Invoke Other Programs

The *IPB* unit has a fixed set of constituent units: *Main*, *PhoneBook*, and *Gui*. But it is often useful to define the shape of a compound unit without immediately specifying all of its constituent units. For example, the interactive phone book can be implemented for different graphical platforms, *e.g.*,

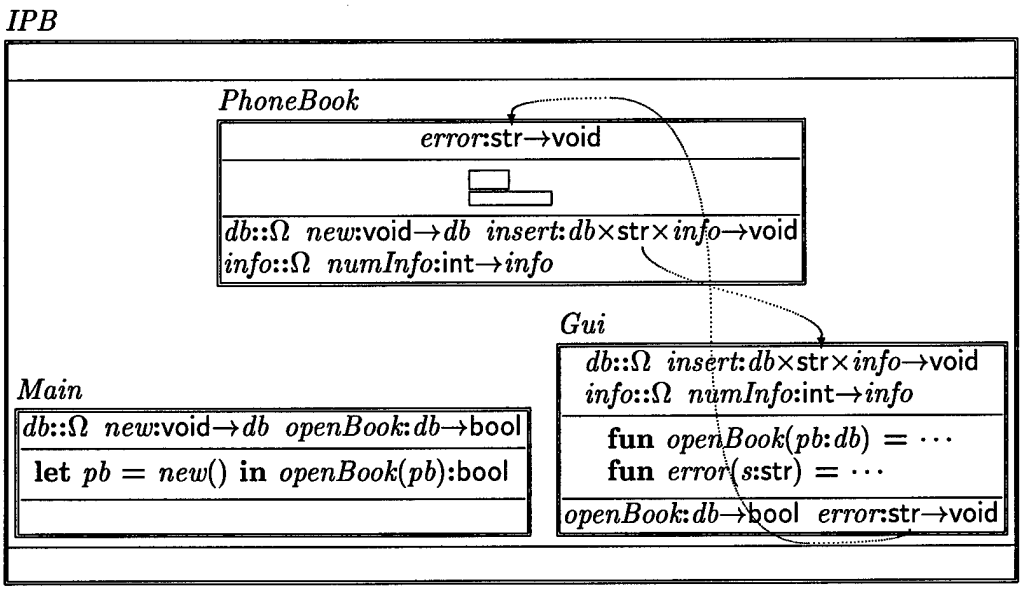


Fig. 4. Cycles in the linking graph are allowed

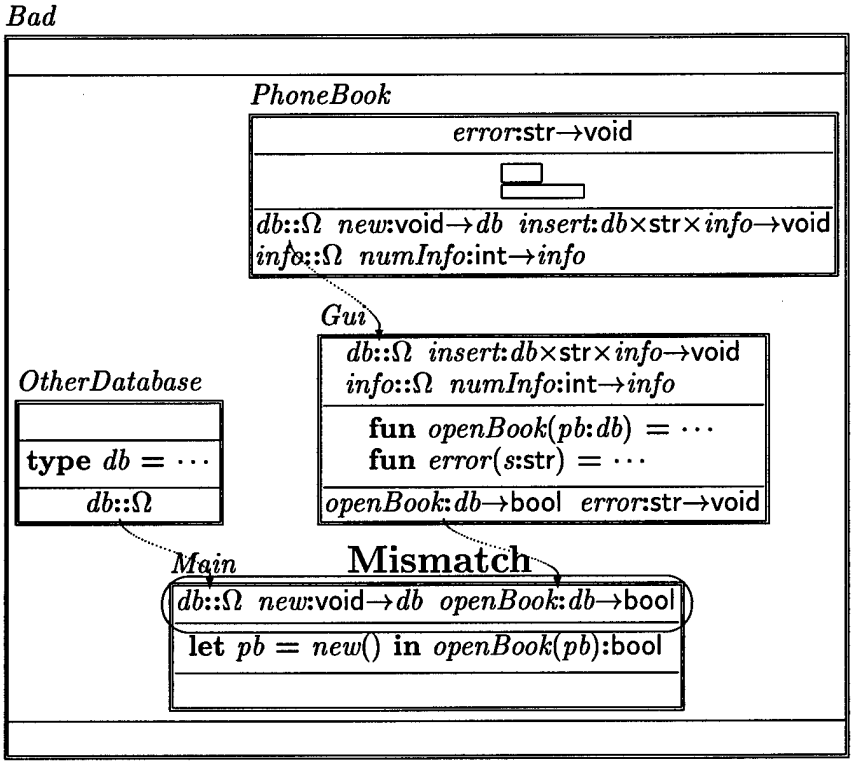


Fig. 5. Illegal linking due to a type mismatch

Macintosh and Windows, by defining different GUI units. Every GUI unit will have the same set of imports and exports, so the linking required to produce the complete interactive phone book is independent of the specific GUI unit. In short, the *IPB* compound unit should be abstracted with respect to its *Gui* unit.

Since units are values, this form of abstraction can be achieved with a function. Figure 6 defines *MakeIPB*, a function that takes a GUI unit and returns an interactive phone book unit. The dashed boxes for *aGui* and *MakeIPB* indicate that the actual GUI and interactive phone book units are not yet determined. *MakeIPB* can be applied to different GUI implementations to produce different interactive phone book programs.

The type associated with *MakeIPB*'s argument is a unit type—a *signature*—that contains all of the information needed to verify its linkage in *MakeIPB*. In the graphical notation, a signature corresponds to a box with imports, exports, and an initialization expression type, but no definitions or expressions. The signature for *aGui* is defined by its dotted box, with **:void** indicating the type of the initialization expression. Using only this signature, the linking specified in *MakeIPB* can be completely verified, and the signature of the resulting compound unit is determined.

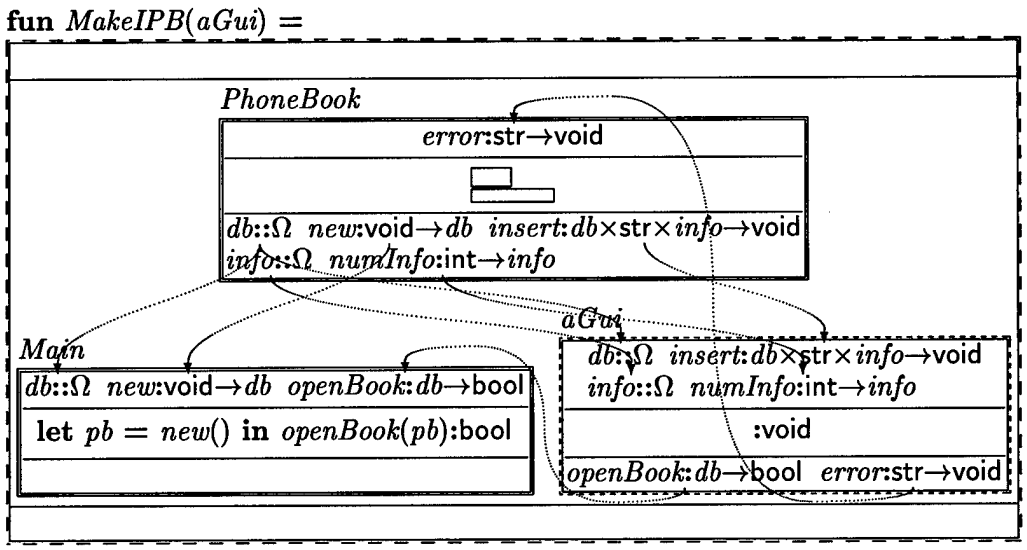


Fig. 6. Abstracting a compound unit over one of its constituent units

The *MakeIPB* function is used to create an interactive phone book, but is not intended to be a function within the interactive phone book program. Instead, *MakeIPB* is part of a linking and invoking program that is written in the same language as the program it links. Invocation is expressed in this language by writing "**invoke**" next to a unit. For example, *Starter* in

Figure 7 is a program that uses **invoke** to run an interactive phone book with a Macintosh GUI.

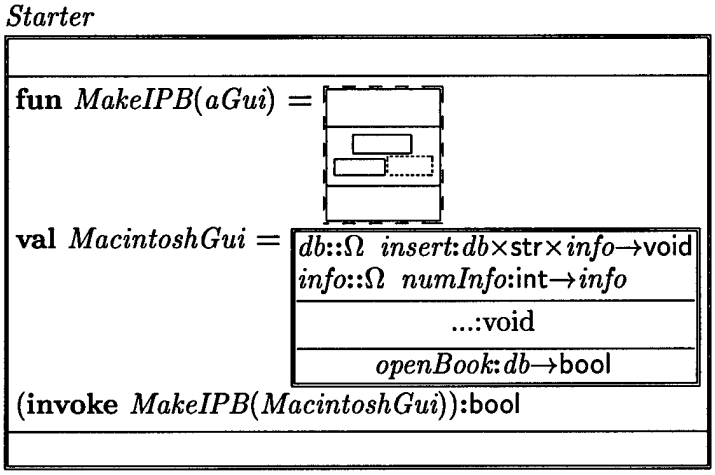


Fig. 7. A program that links and invokes an interactive phone book

2.4 Dynamic Linking

The **invoke** form also works on units that are not complete programs. In this case, the unit's imports are satisfied by types and values from the lexical environment of the **invoke** expression in the invoking program. This generalized form of invocation implements *dynamic linking* for units. For example, the phone book program can exploit dynamic linking to support third-party "plug-in" extensions that load phone numbers from a foreign source. Each loader extension is implemented as a unit that is dynamically linked with the phone book program. The core language must provide a syntactic form that retrieves a unit value from an archive, such as the Internet, and checks that the unit matches a particular signature.[5] Then, a phone book user can install a loader extension at run-time.

Figure 8 defines a *Gui* unit that supports loader extensions. The function *addLoader* consumes a loader extension as a unit and dynamically links it into the program using **invoke**. The extension unit imports types and functions that enable it to modify the phone book database. These imports are satisfied in the **invoke** expression with types and variables that were originally imported into *Gui*, plus the *error* function defined within *Gui*. The result of invoking the extension unit is the value of the unit's initialization expression,

[5] Type-checking in the load expression's context ensures that dynamic linking is type-safe. Java's dynamic class loading is broken because it checks types in a type environment that may differ from the environment where the clas is used [19].

which is required (via signatures) to be a function with the type $db \times file \rightarrow$void. This function is then installed into the interface's table of loader functions.

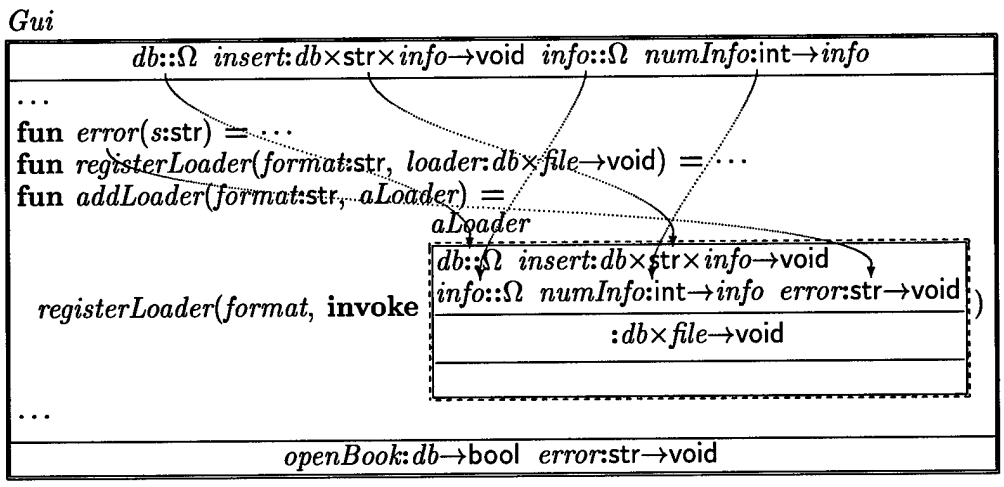


Fig. 8. Dynamic linking with **invoke**

3 The Structure and Interpretation of Units

In this section we develop a precise account of the unit language design in three stages. We start in Section 3.1 with units as an extension of a dynamically typed language to introduce the basic syntax and semantics for units. In Section 3.2, we enrich this language with definitions for constructed types (like classes in Java or datatypes in ML). Finally, in Section 3.3 we consider arbitrary type definitions (like type equations in ML). For all three sections, we only consider those parts of the core language that are immediately relevant to units.

The rigorous description of the unit language, including its type structure and semantics, relies on well-known type checking and rewriting techniques for Scheme and ML [5,11,25]. In the rewriting model of evaluation, the set of program expressions is partitioned into a set of values and a set of non-values. Evaluation is the process of rewriting a non-value expression within a program to an equivalent expression, repeating this process until the whole program is rewritten to a value. For example, a simple unit expression—represented in the graphical language by a box containing text code—is a value, while a compound unit expression is not. A compound unit expression can be rewritten to an equivalent unit expression by merging the text of the constituent units, as demonstrated in Figure 9. Invocation for a unit is similar: an **invoke** expression is rewritten by extracting the invoked unit's definitions and initialization expression, and then replacing references to imported variables with

values supplied for the imports. Otherwise, the standard rules for functions, assignments, and exceptions apply.

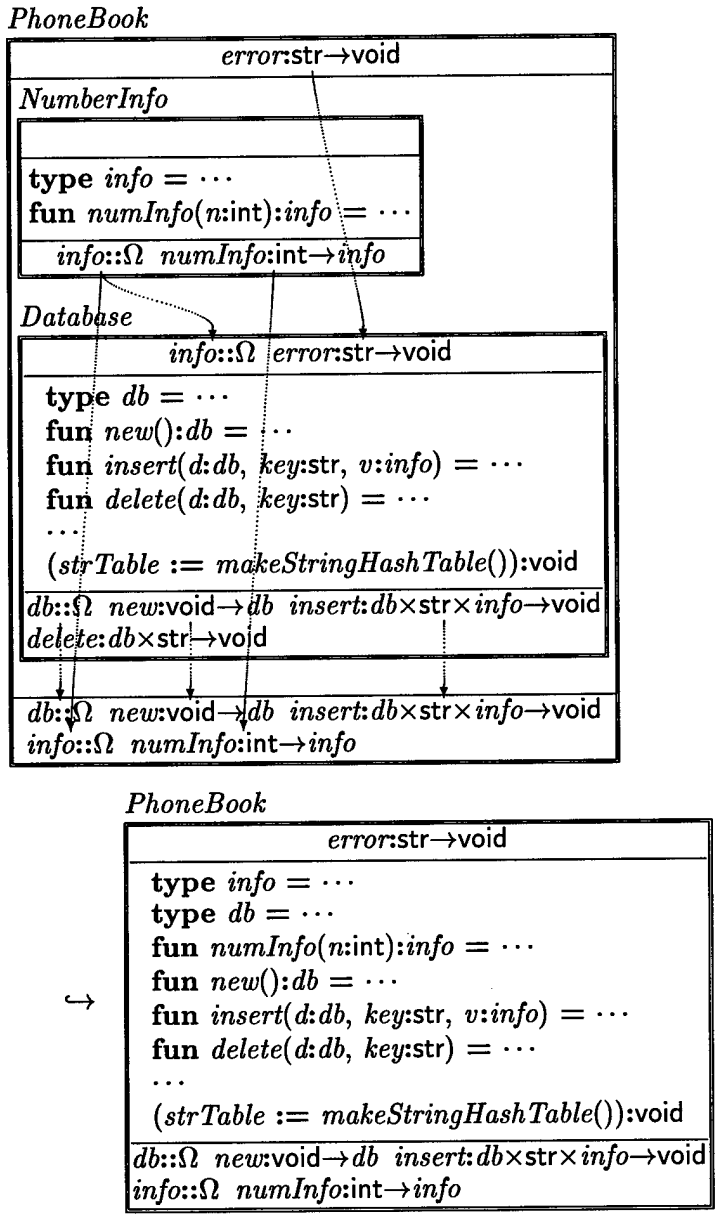


Fig. 9. Graphical reduction rule for a compound unit

3.1 Dynamically Typed Units

Figure 10 defines the syntax of UNIT$_d$, an extension of a dynamically typed core language. The unspecified expression forms of the core language are

extended with three unit-specific forms: a **unit** form for creating units, a **compound** form for linking units into a compound unit, and an **invoke** form for invoking units. The core language must provide two forms that are used in the process of linking and invoking: an expression sequence form (";") and a **letrec** form for lexical blocks containing mututally recusive definitions.

e	=	**unit** *imports exports definitions e*
	|	**compound** *imports exports*
		link *e linkage* **and** *e linkage*
	|	**invoke** *e* **with** *invoke-linkage*
	|	*e* ; *e* | **letrec** *value-defn** **in** *e*
imports	=	**import** *value-var-decl**
exports	=	**export** *value-var-decl**
definitions	=	*value-defn**
value-defn	=	**val** *value-var-decl* = *e*
linkage	=	**with** *value-var-decl** **provides** *value-var-decl**
invoke-linkage	=	*value-invoke-linkage**
value-invoke-linkage	=	*value-var-decl* = *e*
value-var-decl	=	x
x	=	value variable

Fig. 10. Syntax of UNIT_d, units for a dynamically typed core language

The **unit** form consists of a set of import and export declarations followed by internal definitions and an initialization expression. The variables specified in the *imports* section of the unit are bound in the definition and initialization expressions. All variables listed in the *exports* section must be defined within the unit. In each definition, the expression on the right-hand side of = must be a *valuable* expression in the sense of Harper and Stone [11]—*i.e.*, evaluating the expression must not incur any computational effects—with the restriction that imported and defined variables are not considered valuable.[6] The scope of a defined variable includes all of the definition expressions in the unit as well as the initialization expression.

A **unit** expression is a first-class value. There are only two operations on this value: linking the unit and invoking the unit. There is no way to "look inside" of a unit value to extract any information about its definitions or initialization expression. In particular, there is no "dot notation" for accessing parts of a unit, since a unit contains only unevaluated definitions and expressions.

The **compound** form links two constituent units together into a new

[6] This restriction simplifies the presentation of the formal semantics, but it can be lifted for an implementation, as in MzScheme, where accessing an undefined variable is detected as a run-time error.

unit.[7] Like **unit**, the **compound** form starts with a list of imported and exported variables. The imported variables can be supplied as imports to the compound unit's constituents. The exported variables must be a subset of the constituents' exports. The constituent units are determined by two subexpressions: one after the **link** keyword and another after the **and** keyword. Along with each constituent unit expression, the variables that the unit is expected to import are listed after the **with** keyword, and the variables that the unit is expected to export are listed after the **provides** keyword.

Variables are linked within a **compound** unit by name. Thus, the set of variables listed after **with** for the first constituent unit must be a subset of the variables imported by the **compound** expression plus the variables listed after **provides** for the second constiuent unit. Similarly, the variables exported by the **compound** expression must be a subset of the combined set of variables listed after **provides** for each of the constituent units.

A **compound** expression is not a value. It evaluates to a unit value that is indistinguishable from a unit created by **unit** with the same imports and exports. This unit's initialization expression is the sequence of the first constituent unit's initialization expression followed by the the second constiuent unit's.

The **invoke** form consumes a unit, determined by a single expression, and invokes it. If the unit requires any imported values, they can be provided in the *invoke-linkage* section of the **invoke** expression, which associates values with names for the unit's imports. An **invoke** expression evaluates to the invoked unit's initialization expression.

To simplify the presentation, UNIT$_d$ does not allow α-renaming for a unit's imported and exported variables. In MzScheme's implementation of units, imported and exported variables have separate internal and external names, so all bound variables within a unit can be α-renamed. Also, MzScheme's **compound** form links imports and exports via source and destination name pairs, rather than requiring the same name at both ends of a linkage.

3.1.1 UNIT$_d$ *Context-sensitive Checking*

The rules in Figure 11 enforce the context-sensitive properties that were informally described in the previous section. The checks ensure that a variable is not multiply defined, imported, or exported, and that the **link** clause of a **compound** expression is locally consistent.

[7] Linking an arbitrary number of units together in a single **compound** expression (as in MzScheme's implementation) is a simple generalization.

$$\frac{\overline{x}\text{ distinct} \quad \Gamma \vdash e_u \quad \Gamma \vdash \overline{e}}{\Gamma \vdash \textbf{invoke } e_u \textbf{ with } \overline{x = e}}$$

$$\frac{\begin{array}{c}\overline{x_i} \cup \overline{x}\text{ distinct} \quad \overline{x_e} \subseteq \overline{x} \\ \Gamma, \overline{x}, \overline{x_i} \vdash \overline{e} \quad \Gamma, \overline{x}, \overline{x_i} \vdash e_b\end{array}}{\begin{array}{c}\Gamma \vdash \textbf{unit import } \overline{x_i} \textbf{ export } \overline{x_e} \\ \overline{\textbf{val } x = e} \textbf{ in } e_b\end{array}}$$

$$\frac{\begin{array}{c}\overline{x_i} \cup \overline{x_{p1}} \cup \overline{x_{p2}}\text{ distinct} \quad \overline{x_e}\text{ distinct} \\ \overline{x_{w1}} \subseteq \overline{x_i} \cup \overline{x_{p2}} \quad \overline{x_{w2}} \subseteq \overline{x_i} \cup \overline{x_{p1}} \quad \overline{x_e} \subseteq \overline{x_{p1}} \cup \overline{x_{p2}} \quad \Gamma \vdash e_1 \quad \Gamma \vdash e_2\end{array}}{\begin{array}{l}\Gamma \vdash \textbf{compound import } \overline{x_i} \textbf{ export } \overline{x_e} \\ \qquad \textbf{link } e_1 \textbf{ with } \overline{x_{w1}} \textbf{ provides } \overline{x_{p1}} \\ \qquad \textbf{and } e_2 \textbf{ with } \overline{x_{w2}} \textbf{ provides } \overline{x_{p2}}\end{array}}$$

The notation $\overline{x}$ indicates either a set or sequence of variables x, depending on the context. The notation $\overline{\textbf{val } x = e}$ indicates the sequence **val** $x = e$ where each x is taken from the set $\overline{x}$ with a corresponding e from the set $\overline{e}$.

Fig. 11. Checking the form of UNIT$_d$ expressions

$$\begin{array}{l}\textbf{invoke (unit import } \overline{x_i} \textbf{ export } \overline{x_e} \textbf{ if } \overline{x_i} \subseteq \overline{x_w} \\ \qquad\qquad \overline{\textbf{val } x = e} \textbf{ in } e_b) \\ \qquad \textbf{with } \overline{x_w = v_w} \\ \hookrightarrow \quad \overline{[v_w/x_w]}(\textbf{letrec } \overline{\textbf{val } x = e} \textbf{ in } e_b)\end{array}$$

$$\begin{array}{l}\textbf{compound import } \overline{x_i} \textbf{ export } \overline{x_e} \\ \qquad \textbf{link (unit import } \overline{x_{i1}} \textbf{ export } \overline{x_{e1}} \\ \qquad\qquad \overline{\textbf{val } x_1 = e_1} \textbf{ in } e_{b1}) \quad \textbf{with } \overline{x_{w1}} \textbf{ provides } \overline{x_{p1}} \\ \qquad \textbf{and (unit import } \overline{x_{i2}} \textbf{ export } \overline{x_{e2}} \\ \qquad\qquad \overline{\textbf{val } x_2 = e_2} \textbf{ in } e_{b2}) \quad \textbf{with } \overline{x_{w2}} \textbf{ provides } \overline{x_{p2}} \\ \\ \hookrightarrow \quad \textbf{unit import } \overline{x_i} \\ \qquad\qquad \textbf{export } \overline{x_e} \\ \qquad\qquad \overline{\textbf{val } x_1 = e_1} \\ \qquad\qquad \overline{\textbf{val } x_2 = e_2} \\ \qquad\qquad \textbf{in } e_{b1} \textbf{ ; } e_{b2}\end{array}$$

if $\overline{x_1} \cup \overline{x_2} \cup \overline{x_i}$ distinct, $\overline{x_{i1}} \subseteq \overline{x_{w1}}$, $\overline{x_{p1}} \subseteq \overline{x_{e1}}$, $\overline{x_{i2}} \subseteq \overline{x_{w2}}$, and $\overline{x_{p2}} \subseteq \overline{x_{e2}}$

Fig. 12. Reducing UNIT$_d$ expressions

3.1.2 UNIT$_d$ *Evaluation*

The unit-specific reduction rules for UNIT$_d$ are defined in Figure 12. These rules are a modification of those for Scheme [5]. The first rule shows that an **invoke** expression reduces to a **letrec** expression containing the invoked unit's definitions and initialization expression. In this **letrec** expression, imported variables are replaced with values supplied for the imports. The variables supplied by **invoke**'s **with** clause must cover all of the imports required by the unit.

The second rule defines how the **compound** expression combines two units: the definitions from each unit are merged and the initialization expressions are sequenced. The **compound** rule requires that the constituent units provide at least the expected exports (according to the **provides** clauses) and need no more than the expected imports (according to the **with** clauses). The reduction rule also requires that unexported definitions in the two units have been appropriately α-renamed to avoid collisions when the definitions are merged.

3.2 *Units with Constructed Types*

Figure 13 extends the language in Figure 10 for a statically typed language with programmer-defined constructed types, such as ML datatypes. In the new language, UNIT$_c$, the imports and exports of a **unit** expression include type variables as well as value variables. All type variables have a kind[8] and all value variables have a type. The **compound** and **invoke** expressions are extended in the natural way to handle imported and exported types.

The definition section of a **unit** expression contains both type and value definitions. Type definitions of the form **type** $t = x_l\ \tau_l \mid x_r\ \tau_r \triangleright x_s$ are similar to ML **datatype** definitions. For simplicity, every type defined in UNIT$_c$ has exactly two variants. Instances of the first variant are constructed with the x_l function, which takes a value of type τ_l and constructs a value of type t. Instances of the second variant are constructed with x_r given a value of type τ_r. The x_s function is the standard selector function for a datatype.

The type of a **unit** expression is a signature of the form **sig** *imports exports* τ **end** where *imports* specifies the kinds and types of a unit's imports and *exports* describes the kinds and types of its exports. As in **unit**, types in either *imports* or *exports* can be used in the type expressions within the signature. The type expression τ is the type of the unit's initialization expression, which cannot depend on type variables listed in *exports*.

The type checking and evaluation rules for UNIT$_c$ are natural extensions

[8] The only kind in this language is Ω, which is the kind of types for values. We declare explicit kinds in anticipation of future work that handles type constructors and polymorphism, which require kinds such as $\Omega \rightarrow \Omega$.

imports	=	**import** *type-var-decl* value-var-decl**
exports	=	**export** *type-var-decl* value-var-decl**
definitions	=	*datatype-defn* value-defn**
datatype-defn	=	**type** $t = x\ \tau \mid x\ \tau \triangleright x$
linkage	=	**with** *type-var-decl* value-var-decl** **provides** *type-var-decl* value-var-decl**
invoke-linkage	=	*type-invoke-linkage* value-invoke-linkage**
type-invoke-linkage	=	*type-var-decl* $= \tau$
type-var-decl	=	$t :: \kappa$
value-var-decl	=	$x : \tau$
τ, σ	=	$t \mid \tau \to \tau \mid$ *signature*
signature	=	**sig** *imports exports* τ **end**
t	=	type variable
κ	=	type kind

Fig. 13. Syntax extensions for UNIT_c, units for a core language with constructed types

to those of UNIT_d. The interested reader is referred to Appendix A for details.

3.3 Units with Type Equations

UNIT_c is sufficient to extend languages where a new constructor is associated with every defined type. Other languages support type equations of the form **type** $t = \tau$, which defines t as an abbreviation for the type τ. If the complete program is known, the variable t can be replaced everywhere with τ. Otherwise, the expansion of t must be delayed until the program is fully assembled.

UNIT_e extends UNIT_c with type equations. Since two units can contain mutually-recursive definitions, naively linking two units with type equations may result in a cyclic type definition. To prevent cyclic definitions created by linking, signatures in UNIT_e include information about type dependencies.

definitions	=	*type-defn* datatype-defn* value-defn**
type-defn	=	**type** $t :: \kappa = \sigma$
signature	=	**sig** *imports exports* **depends** *dependency** τ **end**
dependency	=	$t \rightsquigarrow t$

Fig. 14. Syntax extensions for UNIT_e, units for a language with type definitions

Figure 14 defines syntax extensions for UNIT_e, including a new signature form that contains a **depends** clause. The dependency declaration $t_e \rightsquigarrow$

t_i means that an exported type t_e depends on an imported type t_i. When two units are linked with a **compound** expression, the unit system traces the set of dependencies to ensure that linking does not create a cyclic type definition. The signature for a **compound** expression propagates dependency information for types imported into and exported from the compound unit.

The type checking and evaluation rules for UNIT_e are natural extensions to those of UNIT_c. The interested reader is referred to Appendix B for details.

4 Implementation

Closed units can be compiled separately in the same way as closed functors in ML. When compiling a unit, imported types are obviously not yet determined and thus have unknown representations. Hence, expressions involving imported types must be compiled like polymorphic functions in ML [22,14]. Otherwise, the restrictions implied by a unit's interface allow inter-procedural optimizations within the unit (such as inlining, specialization, and dead-code elimination). Furthermore, since a compound unit is equivalent to a simple unit that merges its constituent units, *intra*-unit optimization techniques naturally extend to *inter*-unit optimizations when a **compound** expression has known constituent units.

In MzScheme's implementation of UNIT_d, units are compiled by transforming them into procedures. The unit's imported and exported variables are implemented as first-class reference cells that are externally created and passed to the procedure when the unit is invoked. The procedure is responsible for filling the export cells with exported values and for remembering the import cells for accessing imports later. The return value of the procedure is a closure that evaluates the unit's initialization expression. Figure 15 illustrates this transformation on an atomic unit. A compound unit encapsulates a list of constituent units (instead of definitions) and a procedure that propagates import and export cells to the constituent units, creating new cells to implement variables in the constituents that are not exposed by the compound unit.

5 Related Module Languages

Our unit language incorporates ideas that have evolved in distinct language communities:

- Traditional languages like C have relied on the filesystem as the language of modules. Programs (makefiles) manipulate ".o" files to select the modules that are linked into a program, and module files are partially linked to create new ".o" or library files. Modern linking systems such as ELF [21] support dynamic linking. However, even the most advanced linking systems rely on a global namespace of function names and module (*i.e.*, file) names, so that

```
(unit (import even) (export odd)
   (define odd (lambda (x)
                 (if (zero? x)
                     #f
                     (even (sub1 x)))))
   (odd 13))
=>
(lambda (even-cell odd-cell)
  (set-cell! odd-cell
      (lambda (x)
        (if (zero? x)
            #f
            ((cell-value even-cell) (sub1 x)))))
  (lambda () ((cell-value odd-cell) 13)))
```

Fig. 15. An example of the basic compilation strategy for Scheme units

modules can only be linked and invoked once in a program.

- Languages such as Ada [1], Modula-2 [24], Modula-3 [9], Haskell [12], Common Lisp [20], and Java [8] have established the "packages" approach to modularity, in which type and value definitions are grouped into packages that explicitly import parts of other packages. The package system delineates the boundaries of each module and forces the specification of static dependencies between modules. Linking and invocation are clearly separated, which allows mutually recursive function definitions across package boundaries.

  The main weakness of a package system is its reliance on a global namespace of packages with importing connections hardwired into each package. In contrast to our unit language, package systems do not permit the reuse of a single package for multiple invocations in a program or the external selection of connections between packages.[9] There is also no way to merge several packages into a new package that hides parts of the constituent packages. These shortcomings make packages less reuseable than units.

  Among the languages with packages, only Java provides a mechanism for dynamic linking. This mechanism is similar to the dynamic linking for units, but it is expressed indirectly via the language of class loaders, and is not fully general due to the constraints of a global package namespace.

- ML's functor system [17] is the most notable example of a language that lets a programmer describe *abstractions over modules* and gives a programmer direct control over assembling modules. Programmers can create modules that are completely private to other modules by instantiating functors

[9] Modula-3's generics allows the former but not the latter.

anonymously as arguments to other functors. The ML community has produced a large body of work exploring variations on the basic module system, especially variaions for higher-order modules [2,10,15,16,18,23].

Unfortunately, the standard mechanism for combining modules relies prevents the definition of mutually recursive types or procedures across module boundaries. And unlike units, ML provides no mechanism for dynamic linking since the module language is distinct from the evaluation language with a strict phase separation. Duggan and Sourelis have investigated "mixins" as a solution to the recursion problem [4]. Their approach is radically different from ours and does not address the problems of higher-order modules or dynamic linking.

In addition, Cardelli [3] anticipated the unit language's emphasis on module linking as well as module definition. Our unit model is more concrete than his proposal and addresses many of his suggestions for future work. Kelsey's proposed module system for Scheme [13] captures most of the organizational properties of units, but does not address static typing or dynamic linking. In short, our unit model fuses the best parts of existing module systems in a novel, compact way that is applicable to many core languages.

6 Conclusion

Encapsulating program fragments is only half the story for modular programming. The other half is linking and invoking these encapsulations, sometimes in the context of a program that is already executing. A promising approach to giving programmers control over the latter half is to integrate program fragments into the core programming langauge. We have shown how this can be accomplished with program units to give the programmer a flexible language for combining programs fragments without sacrificing the distinct phases of linking and evluation.

The unit language was originally implemented to simplify the development of DrScheme [6], Rice's Scheme programming environment. Units simplify DrScheme's implementation as a large and *dynamic* program. DrScheme supports multiple language dialects and third-party extensions that hook into its complex graphical interface. DrScheme also acts as a kind of operating system for client programs that are being developed, launching client programs by dynamically linking them into the system while maintaing the boundaries between clients. Units express DrScheme's extensibility and OS-nature directly and elegantly.

Our proposal does not necessitate a tight integration of units into the core language. A weaker form of integration—using a separate language for defining and linking units—can acheive similar benefits if essential design features are kept intact: compound units, a mechanism to inject units into the set of run-

time values, and an core expression for invoking units. In future work, we intend to explore linking languages that are separate from the core language to determine the optimal level of integration.

References

[1] BARNES, J. G. P. *Programming in Ada 95.* Addison-Wesley, 1996.

[2] BISWAS, S. K. Higher-order functors with transparent signatures. In *Proc. ACM Symposium on Principles of Programming Languages* (1995), pp. 154–163.

[3] CARDELLI, L. Program fragments, linking, and modularization. In *Proc. ACM Symposium on Principles of Programming Languages* (1997), pp. 266–277.

[4] DUGGAN, D., AND SOURELIS, C. Mixin modules. In *Proc. ACM International Conference on Functional Programming* (1996), pp. 262–273.

[5] FELLEISEN, M., AND HIEB, R. The revised report on the syntactic theories of sequential control and state. Tech. Rep. 100, Rice University, June 1989. *Theoretical Computer Science*, volume 102, 1992, pp. 235–271.

[6] FINDLER, R. B., FLANAGAN, C., FLATT, M., KRISHNAMURTHI, S., AND FELLEISEN, M. DrScheme: A pedagogic programming environment for Scheme. In *Proc. International Symposium on Programming Languages: Implementations, Logics, and Programs* (1997), pp. 369–388.

[7] FLATT, M. PLT MzScheme: Language manual. Tech. Rep. TR97-280, Rice University, 1997.

[8] GOSLING, J., JOY, B., AND STEELE, G. *The Java Language Specification.* The Java Series. Addison-Wesley, Reading, MA, USA, June 1996.

[9] HARBISON, S. P. *Modula-3.* Prentice Hall, 1991.

[10] HARPER, R., AND LILLIBRIDGE, M. A type-theoretic approach to higher-order modules with sharing. In *Proc. ACM Symposium on Principles of Programming Languages* (1994), pp. 123–137.

[11] HARPER, R., AND STONE, C. A type-theoretic semantics for Standard ML 1996. Submitted for publication, 1997.

[12] HUDAK, P., AND WADLER, P. (EDS.). Report on the programming language Haskell. Tech. Rep. YALE/DCS/RR777, Yale University, Department of Computer Science, Aug. 1991.

[13] KELSEY, R. A. Fully-parameterized modules or the missing link. Tech. Rep. 97-3, NEC Research Institute, 1997.

[14] LEROY, X. Unboxed objects and polymorphic typing. In *Proc. ACM Symposium on Principles of Programming Languages* (1992), pp. 177–188.

[15] LEROY, X. Manifest types, modules, and separate compilation. In *Proc. ACM Symposium on Principles of Programming Languages* (1994), pp. 109–122.

[16] LEROY, X. Applicative functions and fully transparent higher-order modules. In *Proc. ACM Symposium on Principles of Programming Languages* (1995), pp. 142–153.

[17] MACQUEEN, D. Modules for Standard ML. In *Proc. ACM Conference on Lisp and Functional Programming* (1984), pp. 198–207.

[18] MACQUEEN, D. B., AND TOFTE, M. A semantics for higher-order functors. In *European Symposium on Programming* (Apr. 1994), Springer-Verlag, LNCS 788, pp. 409–423.

[19] SARASWAT, V. Java is not type-safe, Aug. 1997. URL: `www.research.att.com/~vj/bug.html`.

[20] STEELE JR., G. L. *Common Lisp: The Language*, second ed. Digital Press, 1990.

[21] SUNSOFT. *SunOS 5.5 Linker and Libraries Manual*, 1996.

[22] TARDITI, D., MORRISETT, G., CHENG, P., STONE, C., HARPER, R., AND LEE, P. TIL: A type-directed optimizing compiler for ML. In *Proc. ACM Conference on Programming Language Design and Implementation* (1996), pp. 181–192.

[23] TOFTE, M. Principal signatures for higher-order program modules. In *Proc. ACM Symposium on Principles of Programming Languages* (1992), pp. 189–199.

[24] WIRTH, N. *Programming in Modula-2.* Springer-Verlag, 1983.

[25] WRIGHT, A., AND FELLEISEN, M. A syntactic approach to type soundness. Tech. Rep. 160, Rice University, 1991. *Information and Computation*, volume 115(1), 1994, pp. 38–94.

Appendix

A UNIT$_c$ Type Checking and Evaluation

For economy, we introduce the following unusual abbreviation, which summarizes the content of a signature with the indices used on names:

$$\begin{aligned}\mathsf{sig}[i, e, b] \equiv\ & \textbf{sig import } \overrightarrow{t_i::\kappa_i}\ \overrightarrow{x_i:\tau_i} \\ & \textbf{export } \overrightarrow{t_e::\kappa_e}\ \overrightarrow{x_e:\tau_e} \\ & \tau_b\end{aligned}$$

To allow the use of specialized units in place of more general units, signatures have a subtype relation (see Figure A.1): a specific signature t_s is a subtype of a more general signature t_g if 1) t_s has fewer imports and more exports, 2) the type of each imported name in t_s is a subtype of the one in t_g, 3) the type of each exported name in t_g is a subtype of the one in t_s, and 4) the type of the body expression in t_s is a subtype of the body expression type in t_g.

$$\frac{\begin{array}{c}\tau_{b1} \le \tau_{b2} \quad \overrightarrow{t_{i1}::\kappa_{i1}} \subseteq \overrightarrow{t_{i2}::\kappa_{i2}} \quad \overrightarrow{t_{e1}::\kappa_{e1}} \supseteq \overrightarrow{t_{e2}::\kappa_{e2}} \\ \forall x_{i1}:\tau_{i1} \in \overrightarrow{x_{i1}:\tau_{i1}}, \exists x_{i1}:\tau_{i2} \in \overrightarrow{x_{i2}:\tau_{i2}} \text{ s.t. } \tau_{i2} \le \tau_{i1} \\ \forall x_{e2}:\tau_{e2} \in \overrightarrow{x_{e2}:\tau_{e2}}, \exists x_{e2}:\tau_{e1} \in \overrightarrow{x_{e1}:\tau_{e1}} \text{ s.t. } \tau_{e1} \le \tau_{e2}\end{array}}{\mathsf{sig}[i1, e1, b1] \le \mathsf{sig}[i2, e2, b2]} \qquad \frac{\Gamma \vdash e : \tau' \quad \tau' \le \tau}{\Gamma \vdash_s e : \tau}$$

Fig. A.1. Subtyping and subsumption in UNIT$_c$ signatures

The typing rules for UNIT$_c$ are shown in Figure A.2. These rules are typed extensions of the rules from Section 3.1.1. The special rule $\vdash_s$ is used when subsumption is allowed on an expression's type. Subsumption is used carefully so that type checking is deterministic. For example, full subsumption is not allowed in the expression e_u for the **invoke** rule because the initialization expression type τ_b in e_u's signature supplies the type of the entire **invoke** expression.

The reduction rules for UNIT$_c$ in Figure A.3 resemble the reductions in Section 3.1.2. The only difference for UNIT$_c$ is that the **invoke** and **compound** reductions propagate **type** definitions as well as **val** definitions.

$$\frac{\Gamma' = \Gamma, \overline{t_i :: \kappa_i}, \overline{t_e :: \kappa_e} \quad FTV(\tau_b) \cap \overline{t_e} = \emptyset \quad \Gamma' \vdash \tau_i :: \kappa_i \quad \Gamma' \vdash \tau_e :: \kappa_e \quad \Gamma' \vdash \tau_b :: \Omega}{\Gamma \vdash \mathsf{sig}[i, e, b] :: \Omega}$$

$$\frac{\begin{array}{c}\overline{t} \cup \overline{x} \text{ distinct} \quad \Gamma' \vdash \tau_b :: \Omega \quad \Gamma \vdash \overline{\sigma} :: \overline{\kappa} \\ \Gamma \vdash_{\mathsf{s}} \overline{e} : \overline{\tau} \quad \Gamma \vdash e_u : \mathsf{sig}[i, e, b] \\ \mathsf{sig}[i, e, b] \leq \textbf{sig import } \overline{t :: \kappa}\ \overline{x : \tau} \textbf{ export } \tau_b\end{array}}{\Gamma \vdash \textbf{invoke } e_u \textbf{ with } \overline{t :: \kappa = \sigma}\ \overline{x : \tau = e} : \tau_b}$$

$$\frac{\begin{array}{c}\overline{t_i} \cup \overline{t} \cup \overline{x_i} \cup \overline{x_{\mathsf{l}}} \cup \overline{x_{\mathsf{r}}} \cup \overline{x_{\mathsf{s}}} \cup \overline{x} \text{ distinct} \quad \overline{t_e :: \kappa_e} \cup \overline{x_e : \tau_e} \subseteq \overline{t :: \Omega} \cup \overline{x : \tau} \cup \overline{x_{\mathsf{l}} : \tau_{\mathsf{l}}} \cup \overline{x_{\mathsf{r}} : \tau_{\mathsf{r}}} \cup \overline{x_{\mathsf{s}} : \tau_{\mathsf{s}}} \\ \Gamma \vdash \mathsf{sig}[i, e, b] :: \Omega \quad \Gamma' = \Gamma, \overline{t_i :: \kappa_i}, \overline{t :: \Omega} \quad \Gamma' \vdash \overline{\tau_{\mathsf{l}}} :: \Omega \quad \Gamma' \vdash \overline{\tau_{\mathsf{r}}} :: \Omega \quad \Gamma' \vdash \overline{\tau} :: \Omega \\ \Gamma'' = \Gamma', \overline{x_i : \tau_i}, \overline{x_{\mathsf{l}} : \tau_{\mathsf{l}} \to t}, \overline{x_{\mathsf{r}} : \tau_{\mathsf{r}} \to t}, \overline{x_{\mathsf{s}} : t \to (\tau_{\mathsf{l}} \to \alpha) \to (\tau_{\mathsf{r}} \to \alpha) \to \alpha} \\ \Gamma'' \vdash_{\mathsf{s}} \overline{e} : \overline{\tau} \quad \Gamma'' \vdash e_b : \tau_b\end{array}}{\begin{array}{l}\Gamma \vdash \textbf{unit import } \overline{t_i :: \kappa_i}\ \overline{x_i : \tau_i} \quad \textbf{export } \overline{t_e :: \kappa_e}\ \overline{x_e : \tau_e} \\ \qquad \overline{\textbf{type } t = x_{\mathsf{l}}\ \tau_{\mathsf{l}} \mid x_{\mathsf{r}}\ \tau_{\mathsf{r}} \triangleright x_{\mathsf{s}}} \\ \qquad \overline{\textbf{val } x : \tau = e} \textbf{ in } e_b \\ \quad : \mathsf{sig}[i, e, b]\end{array}}$$

$$\frac{\begin{array}{c}\overline{t_i} \cup \overline{t_{p1}} \cup \overline{t_{p2}} \cup \overline{x_i} \cup \overline{x_{p1}} \cup \overline{x_{p2}} \text{ distinct} \quad \overline{t_e} \cup \overline{x_e} \text{ distinct} \\ \overline{t_{w1} :: \kappa_{w1}} \cup \overline{x_{w1} : \tau_{w1}} \subseteq \overline{t_i :: \kappa_i} \cup \overline{t_{p2} :: \kappa_{p2}} \cup \overline{x_i : \tau_i} \cup \overline{x_{p2} : \tau_{p2}} \\ \overline{t_{w2} :: \kappa_{w2}} \cup \overline{x_{w2} : \tau_{w2}} \subseteq \overline{t_i :: \kappa_i} \cup \overline{t_{p1} :: \kappa_{p1}} \cup \overline{x_i : \tau_i} \cup \overline{x_{p1} : \tau_{p1}} \\ \overline{t_e :: \kappa_e} \cup \overline{x_e : \tau_e} \subseteq \overline{t_{p1} :: \kappa_{p1}} \cup \overline{t_{p2} :: \kappa_{p2}} \cup \overline{x_{p1} : \tau_{p1}} \cup \overline{x_{p2} : \tau_{p2}} \\ \Gamma \vdash \mathsf{sig}[i, e, b2] :: \Omega \quad \Gamma \vdash \mathsf{sig}[w1, p1, b1] :: \Omega \quad \Gamma \vdash \mathsf{sig}[w2, p2, b2] :: \Omega \\ \Gamma \vdash e_1 : \mathsf{sig}[i1, e1, b1] \quad \Gamma \vdash e_2 : \mathsf{sig}[i2, e2, b2] \\ \mathsf{sig}[i1, e1, b1] \leq \mathsf{sig}[w1, p1, b1] \quad \mathsf{sig}[i2, e2, b2] \leq \mathsf{sig}[w2, p2, b2]\end{array}}{\begin{array}{l}\Gamma \vdash \textbf{compound import } \overline{t_i :: \kappa_i}\ \overline{x_i : \tau_i} \textbf{ export } \overline{t_e :: \kappa_e}\ \overline{x_e : \tau_e} \\ \qquad \textbf{link } e_1 \textbf{ with } \overline{t_{w1} :: \kappa_{w1}}\ \overline{x_{w1} : \tau_{w1}} \textbf{ provides } \overline{t_{p1} :: \kappa_{p1}}\ \overline{x_{p1} : \tau_{p1}} \\ \qquad \textbf{and } e_2 \textbf{ with } \overline{t_{w2} :: \kappa_{w2}}\ \overline{x_{w2} : \tau_{w2}} \textbf{ provides } \overline{t_{p2} :: \kappa_{p2}}\ \overline{x_{p2} : \tau_{p2}} \\ \quad : \mathsf{sig}[i, e, b2]\end{array}}$$

Fig. A.2. Type checking for UNIT$_{\mathsf{c}}$

B UNIT$_{\mathsf{e}}$ Type Checking and Evaluation

The following abbreviation expresses a UNIT$_{\mathsf{e}}$ signature:

$$\begin{array}{rl}\mathsf{sig}[i, e, di, de, b] \equiv & \textbf{sig import } \overline{t_i :: \kappa_i}\ \overline{x_i : \tau_i} \\ & \textbf{export } \overline{t_e :: \kappa_e}\ \overline{x_e : \tau_e} \\ & \textbf{depends } \overline{t_{de} \rightsquigarrow t_{di}} \\ & \tau_b\end{array}$$

$$
\begin{array}{l}
\textbf{invoke}\ (\textbf{unit}\ \textbf{import}\ \overrightarrow{t_i::\kappa_i}\ \overrightarrow{x_i:\tau_i}\ \ \textbf{export}\ \overrightarrow{t_e::\kappa_e}\ \overrightarrow{x_e:\tau_e} \\
\qquad\qquad \overrightarrow{\textbf{type}\ t = x_{\mathsf{l}}\ \tau_{\mathsf{l}} \mid x_{\mathsf{r}}\ \tau_{\mathsf{r}} \triangleright x_{\mathsf{s}}} \\
\qquad\qquad \overrightarrow{\textbf{val}\ x{:}\tau = e}\ \textbf{in}\ e_b) \\
\qquad\ \textbf{with}\ \overrightarrow{t_w::\kappa_w = \sigma_w}\ \overrightarrow{x_w{:}\tau_w = v_w} \\
\\
\hookrightarrow\ \ [\overrightarrow{\tau_w/\sigma_w}, \overrightarrow{v_w/x_w}](\textbf{letrec}\ \overrightarrow{\textbf{type}\ t = x_{\mathsf{l}}\ \tau_{\mathsf{l}} \mid x_{\mathsf{r}}\ \tau_{\mathsf{r}} \triangleright x_{\mathsf{s}}} \\
\qquad\qquad\qquad\qquad \overrightarrow{\textbf{val}\ x{:}\tau = e}\ \textbf{in}\ e_b)
\end{array}
$$

$$
\begin{array}{l}
\textbf{compound import}\ \overrightarrow{t_i::\kappa_i}\ \overrightarrow{x_i:\tau_i}\ \textbf{export}\ \overrightarrow{t_e::\kappa_e}\ \overrightarrow{x_e:\tau_e} \\
\qquad \textbf{link}\ (\textbf{unit}\ \textbf{import}\ \overrightarrow{t_{i1}::\kappa_{i1}}\ \overrightarrow{x_{i1}:\tau_{i1}}\ \textbf{export}\ \overrightarrow{t_{e1}::\kappa_{e1}}\ \overrightarrow{x_{e1}:\tau_{e1}} \\
\qquad\qquad \overrightarrow{\textbf{type}\ t_1 = x_{\mathsf{l}1}\ \tau_{\mathsf{l}1} \mid x_{\mathsf{r}1}\ \tau_{\mathsf{r}1} \triangleright x_{\mathsf{s}1}} \\
\qquad\qquad \overrightarrow{\textbf{val}\ x_1{:}\tau_1 = e_1} \\
\qquad\qquad \textbf{in}\ e_{b1}) \\
\qquad\quad \textbf{with}\ \overrightarrow{t_{w1}::\kappa_{w1}}\ \overrightarrow{x_{w1}:\tau_{w1}} \\
\qquad\quad \textbf{provides}\ \overrightarrow{t_{p1}::\kappa_{p1}}\ \overrightarrow{x_{p1}:\tau_{p1}} \\
\qquad \textbf{and}\ (\textbf{unit}\ \textbf{import}\ \overrightarrow{t_{i2}::\kappa_{i2}}\ \overrightarrow{x_{i2}:\tau_{i2}}\ \textbf{export}\ \overrightarrow{t_{e2}::\kappa_{e2}}\ \overrightarrow{x_{e2}:\tau_{e2}} \\
\qquad\qquad \overrightarrow{\textbf{type}\ t_2 = x_{\mathsf{l}2}\ \tau_{\mathsf{l}2} \mid x_{\mathsf{r}2}\ \tau_{\mathsf{r}2} \triangleright x_{\mathsf{s}2}} \\
\qquad\qquad \overrightarrow{\textbf{val}\ x_2{:}\tau_2 = e_2} \\
\qquad\qquad \textbf{in}\ e_{b2}) \\
\qquad\quad \textbf{with}\ \overrightarrow{t_{w2}::\kappa_{w2}}\ \overrightarrow{x_{w2}:\tau_{w2}} \\
\qquad\quad \textbf{provides}\ \overrightarrow{t_{p2}::\kappa_{p2}}\ \overrightarrow{x_{p2}:\tau_{p2}} \\
\\
\hookrightarrow\ \ \textbf{unit}\ \textbf{import}\ \overrightarrow{t_i::\kappa_i}\ \overrightarrow{x_i:\tau_i} \\
\qquad\quad \textbf{export}\ \overrightarrow{t_e::\kappa_e}\ \overrightarrow{x_e:\tau_e} \\
\qquad\quad \overrightarrow{\textbf{type}\ t_1 = x_{\mathsf{l}1}\ \tau_{\mathsf{l}1} \mid x_{\mathsf{r}1}\ \tau_{\mathsf{r}1} \triangleright x_{\mathsf{s}1}} \\
\qquad\quad \overrightarrow{\textbf{type}\ t_2 = x_{\mathsf{l}2}\ \tau_{\mathsf{l}2} \mid x_{\mathsf{r}2}\ \tau_{\mathsf{r}2} \triangleright x_{\mathsf{s}2}} \\
\qquad\quad \overrightarrow{\textbf{val}\ x_1{:}\tau_1 = e_1} \\
\qquad\quad \overrightarrow{\textbf{val}\ x_2{:}\tau_2 = e_2} \\
\qquad\quad \textbf{in}\ e_{b1}\ ;\ e_{b2}
\end{array}
$$

if $\overrightarrow{t_1} \cup \overrightarrow{t_2} \cup \overrightarrow{t_i} \cup \overrightarrow{x_1} \cup \overrightarrow{x_2} \cup \overrightarrow{x_i}$ distinct

Fig. A.3. Reduction rules for UNIT_{c}

The subtyping rule in Figure B.1 accounts for the new dependency declarations by requiring that a specific signature declares more dependencies than a more general signature.

The type checking rules for UNIT_{e} are defined in Figure B.2. To calculate type equation dependencies for the signature of a simple unit, the type checking rules rely on the $\propto_D$ relation, which associates a type expression with each

$$\frac{\begin{array}{c}\tau_{b1} \leq \tau_{b2} \\ \overrightarrow{t_{i1}::\kappa_{i1}} \subseteq \overrightarrow{t_{i2}::\kappa_{i2}} \quad \overrightarrow{t_{e1}::\kappa_{e1}} \supseteq \overrightarrow{t_{e2}::\kappa_{e2}} \\ \overrightarrow{t_{de1} \rightsquigarrow t_{di1}} \subseteq \overrightarrow{t_{de2} \rightsquigarrow t_{di2}} \\ \forall x_{i1}{:}\tau_{i1} \in \overrightarrow{x_{i1}{:}\tau_{i1}}, \exists x_{i1}{:}\tau_{i2} \in \overrightarrow{x_{i2}{:}\tau_{i2}} \text{ s.t. } \tau_{i2} \leq \tau_{i1} \\ \forall x_{e2}{:}\tau_{e2} \in \overrightarrow{x_{e2}{:}\tau_{e2}}, \exists x_{e2}{:}\tau_{e1} \in \overrightarrow{x_{e1}{:}\tau_{e1}} \text{ s.t. } \tau_{e1} \leq \tau_{e2}\end{array}}{\mathsf{sig}[i1, e1, di1, de1, b1] \leq \mathsf{sig}[i2, e2, di2, de2, b2]}$$

Fig. B.1. Subtyping and subsumption in UNIT$_e$ signatures

of the type variables it references from the set of type equations D:

$$\begin{aligned}\tau \propto_D t \text{ iff } & t \in FTV(\tau) \\ & \text{or } (\exists \langle t' = \tau' \rangle \in D \text{ s.t. } t' \in FTV(\tau) \text{ and } \tau' \propto_D t)\end{aligned}$$

$FTV(\tau)$ denotes the set of type variables in τ that are not bound by the **import** or **export** clause of a **sig** type. Types in a set of type equations D can be eliminated from a type expression with the $|\bullet|_D$ operator, as follows:

$$|\tau|_D = \begin{cases} t & \text{if } \tau{=}t \text{ and } t \notin D \\ |\tau'|_D & \text{if } \tau{=}t \text{ and } \langle t = \tau' \rangle \in D \\ |\tau'|_D \rightarrow |\tau''|_D & \text{if } \tau{=}\tau' \rightarrow \tau'' \\ \textbf{sig import } \overrightarrow{t_i::\kappa_i}\ \overrightarrow{x_i{:}|\tau_i|_{D'}} & \text{if } \tau{=}\mathsf{sig}[i, e, di, de, b] \\ \quad \textbf{export } \overrightarrow{t_e::\kappa_e}\ \overrightarrow{x_e{:}|\tau_e|_{D'}} & \quad \text{and } D' = \{\langle t = \tau \rangle | \langle t = \tau \rangle \in D \\ \quad \textbf{depends } \overrightarrow{t_{de} \rightsquigarrow t_{di}} & \qquad \text{and } t \notin \overrightarrow{t_i} \cup \overrightarrow{t_e}\} \\ \quad |\tau_b|_{D'} & \end{cases}$$

and similarly expanded in a value expression, sketched as follows:

$$|e|_D = \begin{cases} x & \text{if } e{=}x \\ \textbf{unit} & \text{if } e{=}\textbf{unit} \\ \textbf{import } \overrightarrow{t_i::\kappa_i}\ \overrightarrow{x_i{:}\tau_i} & \quad \textbf{import } \overrightarrow{t_i::\kappa_i}\ \overrightarrow{x_i{:}\tau_i} \\ \textbf{export } \overrightarrow{t_e::\kappa_e}\ \overrightarrow{x_e{:}\tau_e} & \quad \textbf{export } \overrightarrow{t_e::\kappa_e}\ \overrightarrow{x_e{:}\tau_e} \\ \overrightarrow{\textbf{type } t_a::\kappa_a = |\tau_a|_{D'}} & \quad \overrightarrow{\textbf{type } t_a::\kappa_a = \tau_a} \\ \overrightarrow{\textbf{type } t = x_{\mathsf{l}}\ |\tau_{\mathsf{l}}|_{D'} \mid x_{\mathsf{r}}\ |\tau_{\mathsf{r}}|_{D'} \triangleright x_{\mathsf{s}}} & \quad \overrightarrow{\textbf{type } t = x_{\mathsf{l}}\ \tau_{\mathsf{l}} \mid x_{\mathsf{r}}\ \tau_{\mathsf{r}} \triangleright x_{\mathsf{s}}} \\ \overrightarrow{\textbf{val } x{:}|\tau|_{D'} = |e|_{D'}}\ \textbf{in } |e_b|_{D'} & \quad \overrightarrow{\textbf{val } x{:}\tau = e}\ \textbf{in } e_b \\ & \text{and } D' = \{\langle t = \tau \rangle | \langle t = \tau \rangle \in D \\ & \qquad \text{and } t \notin \overrightarrow{t_i} \cup \overrightarrow{t_e} \cup \overrightarrow{t_a} \cup \overrightarrow{t}\} \\ \ldots & \end{cases}$$

The subscript D is left off of $|\bullet|$ when D is clear from context.

The UNIT$_e$ reductions in Figure B.3 differ only slightly from the UNIT$_c$ reductions. Type abbreviations are immediately expanded away in the **invoke**

$$\frac{\begin{array}{c}\Gamma' = \Gamma, \overline{t_i::\kappa_i}, \overline{t_e::\kappa_e} \quad FTV(\tau_b) \cap \overline{t_e} = \emptyset \\ \Gamma' \vdash \tau_i :: \kappa_i \quad \Gamma' \vdash \tau_e :: \kappa_e \quad \Gamma' \vdash \tau_b :: \Omega \\ \overline{t_{de}} \subseteq \overline{t_e} \quad \overline{t_{di}} \subseteq \overline{t_i}\end{array}}{\Gamma \vdash \mathsf{sig}[i, e, di, de, b] \;\; :: \Omega}$$

$$\frac{\begin{array}{c}\overline{t_i} \cup \overline{t} \cup \overline{x_i} \cup \overline{x_\mathsf{l}} \cup \overline{x_\mathsf{r}} \cup \overline{x_\mathsf{s}} \cup \overline{x} \text{ distinct} \quad \overline{t_e::\kappa_e} \cup \overline{x_e:\tau_e} \subseteq \overline{t_a::\kappa_a} \cup \overline{t::\Omega} \cup \overline{x:\tau} \cup \overline{x_\mathsf{l}:\tau_\mathsf{l}} \cup \overline{x_\mathsf{r}:\tau_\mathsf{r}} \cup \overline{x_\mathsf{s}:\tau_\mathsf{s}} \\ D = \overline{\langle t_a = \tau_a \rangle} \quad \tau_a \propto_D t'_a \Rightarrow \tau'_a \not\propto_D t_a \text{ for } \langle t_a = \tau_a \rangle, \langle t'_a = \tau'_a \rangle \in D \\ \overline{t_{de} \rightsquigarrow t_{di}} = \{t_a \rightsquigarrow t_i \mid \langle t_a = \tau_a \rangle \in D \text{ and } t_i \in \overline{t_i} \text{ and } t_a \in \overline{t_e} \text{ and } \tau_a \propto_D t_i\} \\ \Gamma \vdash \mathsf{sig}[i, e, di, de, b] \;\; :: \Omega \quad \Gamma' = \Gamma, \overline{t_i::\kappa_i}, \overline{t::\Omega} \quad \Gamma'_a = \Gamma', \overline{t_a::\kappa_a} \quad \Gamma'_a \vdash \tau_a :: \kappa_a \\ \Gamma' \vdash |\overline{\tau_\mathsf{l}}| :: \Omega \quad \Gamma' \vdash |\overline{\tau_\mathsf{r}}| :: \Omega \quad \Gamma' \vdash |\overline{\tau}| :: \Omega \\ \Gamma'' = \Gamma', \overline{x_i:|\tau_i|}, \overline{x_\mathsf{l}:|\tau_\mathsf{l}| \to |t|}, \overline{x_\mathsf{r}:|\tau_\mathsf{r}| \to |t|}, \overline{x_\mathsf{s}:t \to (|\tau_\mathsf{l}| \to \alpha) \to (|\tau_\mathsf{r}| \to \alpha) \to \alpha} \\ \Gamma'' \vdash_\mathsf{s} |\overline{e}| : |\overline{\tau}| \quad \Gamma'' \vdash |e_b| : \tau_b\end{array}}{\begin{array}{l}\Gamma \vdash \textbf{unit import } \overline{t_i::\kappa_i}\ \overline{x_i:\tau_i} \\ \qquad \textbf{export } \overline{t_e::\kappa_e}\ \overline{x_e:\tau_e} \\ \qquad \overline{\textbf{type } t_a::\kappa_a = \tau_a} \\ \qquad \overline{\textbf{type } t = x_\mathsf{l}\ \tau_\mathsf{l} \mid x_\mathsf{r}\ \tau_\mathsf{r} \triangleright x_\mathsf{s}} \\ \qquad \overline{\textbf{val } x:\tau = e} \textbf{ in } e_b \\ \quad : \mathsf{sig}[i, e, di, de, b]\end{array}}$$

$$\frac{\begin{array}{c}\overline{t_i} \cup \overline{t_{p1}} \cup \overline{t_{p2}} \cup \overline{x_i} \cup \overline{x_{p1}} \cup \overline{x_{p2}} \text{ distinct} \quad \overline{t_e} \cup \overline{x_e} \text{ distinct} \\ \overline{t_{w1}::\kappa_{w1}} \cup \overline{x_{w1}:\tau_{w1}} \subseteq \overline{t_i::\kappa_i} \cup \overline{t_{p2}::\kappa_{p2}} \cup \overline{x_i:\tau_i} \cup \overline{x_{p2}:\tau_{p2}} \\ \overline{t_{w2}::\kappa_{w2}} \cup \overline{x_{w2}:\tau_{w2}} \subseteq \overline{t_i::\kappa_i} \cup \overline{t_{p1}::\kappa_{p1}} \cup \overline{x_i:\tau_i} \cup \overline{x_{p1}:\tau_{p1}} \\ \overline{t_e::\kappa_e} \cup \overline{x_e:\tau_e} \subseteq \overline{t_{p1}::\kappa_{p1}} \cup \overline{t_{p2}::\kappa_{p2}} \cup \overline{x_{p1}:\tau_{p1}} \cup \overline{x_{p2}:\tau_{p2}} \\ \Gamma \vdash \mathsf{sig}[i, e, di, de, b2] :: \Omega \\ \Gamma \vdash \mathsf{sig}[w1, p1, di1, de1, b1] :: \Omega \quad \Gamma \vdash \mathsf{sig}[w2, p2, di2, de2, b2] :: \Omega \\ \Gamma \vdash e_1 : \mathsf{sig}[i1, e1, di1, de1, b1] \quad \Gamma \vdash e_2 : \mathsf{sig}[i2, e2, di2, de2, b2] \\ \mathsf{sig}[i1, e1, di1, de1, b1] \leq \mathsf{sig}[w1, p1, di1, de1, b1] \\ \mathsf{sig}[i2, e2, di2, de2, b2] \leq \mathsf{sig}[w2, p2, di2, de2, b2] \\ \overline{\langle t_{di1}, t_{de1} \rangle} \cap \overline{\langle t_{de2}, t_{di2} \rangle} = \emptyset \\ \overline{t_{de} \rightsquigarrow t_{di}} = \{t_e \rightsquigarrow t_i \mid t_i \in \overline{t_i} \text{ and } t_e \in \overline{t_e} \text{ and } t_e \rightsquigarrow t_i \in \overline{t_{de1} \rightsquigarrow t_{di1}} \cup \overline{t_{de2} \rightsquigarrow t_{di2}}\}\end{array}}{\begin{array}{l}\Gamma \vdash \textbf{compound import } \overline{t_i::\kappa_i}\ \overline{x_i:\tau_i} \textbf{ export } \overline{t_e::\kappa_e}\ \overline{x_e:\tau_e} \\ \qquad \textbf{link } e_1 \textbf{ with } \overline{t_{w1}::\kappa_{w1}}\ \overline{x_{w1}:\tau_{w1}} \textbf{ provides } \overline{t_{p1}::\kappa_{p1}}\ \overline{x_{p1}:\tau_{p1}} \\ \qquad \textbf{and } e_2 \textbf{ with } \overline{t_{w2}::\kappa_{w2}}\ \overline{x_{w2}:\tau_{w2}} \textbf{ provides } \overline{t_{p2}::\kappa_{p2}}\ \overline{x_{p2}:\tau_{p2}} \\ \quad : \mathsf{sig}[i, e, di, de, b2]\end{array}}$$

Fig. B.2. Type checking for UNIT$_\mathsf{e}$

reduction, and the **compound** reduction preserves and merges type equations when linking.

$$
\begin{array}{l}
\textbf{invoke}\ (\textbf{unit}\ \textbf{import}\ \overrightarrow{t_i::\kappa_i}\ \overrightarrow{x_i:\tau_i}\ \ \textbf{export}\ \overrightarrow{t_e::\kappa_e}\ \overrightarrow{x_e:\tau_e} \\
\qquad\qquad \overrightarrow{\textbf{type}\ t_a::\kappa_a = \tau_a} \\
\qquad\qquad \overrightarrow{\textbf{type}\ t = x_{\mathsf{l}}\ \tau_{\mathsf{l}} \mid x_{\mathsf{r}}\ \tau_{\mathsf{r}} \triangleright x_{\mathsf{s}}} \\
\qquad\qquad \overrightarrow{\textbf{val}\ x{:}\tau = e}\ \textbf{in}\ e_b) \\
\qquad \textbf{with}\ \overrightarrow{t_w::\kappa_w = \sigma_w}\ \overrightarrow{x_w{:}\tau_w = v_w} \\
\\
\hookrightarrow\ \ [\overrightarrow{\tau_w/\sigma_w}, \overrightarrow{v_w/x_w}](\textbf{letrec}\ \overrightarrow{\textbf{type}\ t = x_{\mathsf{l}}\ |\tau_{\mathsf{l}}| \mid x_{\mathsf{r}}\ |\tau_{\mathsf{r}}| \triangleright x_{\mathsf{s}}} \\
\qquad\qquad\qquad\qquad\qquad \overrightarrow{\textbf{val}\ x{:}|\tau| = |e|}\ \textbf{in}\ |e_b|)
\end{array}
$$

where $D = \overrightarrow{\langle t_a = \tau_a \rangle}$

$$
\begin{array}{l}
\textbf{compound}\ \textbf{import}\ \overrightarrow{t_i::\kappa_i}\ \overrightarrow{x_i:\tau_i}\ \textbf{export}\ \overrightarrow{t_e::\kappa_e}\ \overrightarrow{x_e:\tau_e} \\
\qquad \textbf{link}\ (\textbf{unit}\ \textbf{import}\ \overrightarrow{t_{i1}::\kappa_{i1}}\ \overrightarrow{x_{i1}:\tau_{i1}}\ \textbf{export}\ \overrightarrow{t_{e1}::\kappa_{e1}}\ \overrightarrow{x_{e1}:\tau_{e1}} \\
\qquad\qquad \overrightarrow{\textbf{type}\ t_{a1}::\kappa_{a1} = \tau_{a1}} \\
\qquad\qquad \overrightarrow{\textbf{type}\ t_1 = x_{\mathsf{l}1}\ \tau_{\mathsf{l}1} \mid x_{\mathsf{r}1}\ \tau_{\mathsf{r}1} \triangleright x_{\mathsf{s}1}} \\
\qquad\qquad \overrightarrow{\textbf{val}\ x_1{:}\tau_1 = e_1}\ \textbf{in}\ e_{b1}) \\
\qquad\qquad \textbf{with}\ \overrightarrow{t_{w1}::\kappa_{w1}}\ \overrightarrow{x_{w1}:\tau_{w1}} \\
\qquad\qquad \textbf{provides}\ \overrightarrow{t_{p1}::\kappa_{p1}}\ \overrightarrow{x_{p1}:\tau_{p1}} \\
\qquad \textbf{and}\ (\textbf{unit}\ \textbf{import}\ \overrightarrow{t_{i2}::\kappa_{i2}}\ \overrightarrow{x_{i2}:\tau_{i2}}\ \textbf{export}\ \overrightarrow{t_{e2}::\kappa_{e2}}\ \overrightarrow{x_{e2}:\tau_{e2}} \\
\qquad\qquad \overrightarrow{\textbf{type}\ t_{a2}::\kappa_{a2} = \tau_{a2}} \\
\qquad\qquad \overrightarrow{\textbf{type}\ t_2 = x_{\mathsf{l}2}\ \tau_{\mathsf{l}2} \mid x_{\mathsf{r}2}\ \tau_{\mathsf{r}2} \triangleright x_{\mathsf{s}2}} \\
\qquad\qquad \overrightarrow{\textbf{val}\ x_2{:}\tau_2 = e_2}\ \textbf{in}\ e_{b2}) \\
\qquad\qquad \textbf{with}\ \overrightarrow{t_{w2}::\kappa_{w2}}\ \overrightarrow{x_{w2}:\tau_{w2}} \\
\qquad\qquad \textbf{provides}\ \overrightarrow{t_{p2}::\kappa_{p2}}\ \overrightarrow{x_{p2}:\tau_{p2}} \\
\\
\hookrightarrow\ \ \textbf{unit}\ \textbf{import}\ \overrightarrow{t_i::\kappa_i}\ \overrightarrow{x_i:\tau_i} \\
\qquad\qquad \textbf{export}\ \overrightarrow{t_e::\kappa_e}\ \overrightarrow{x_e:\tau_e} \\
\qquad\qquad \overrightarrow{\textbf{type}\ t_{a1}::\kappa_{a1} = \tau_{a1}} \\
\qquad\qquad \overrightarrow{\textbf{type}\ t_{a2}::\kappa_{a2} = \tau_{a2}} \\
\qquad\qquad \overrightarrow{\textbf{type}\ t_1 = x_{\mathsf{l}1}\ \tau_{\mathsf{l}1} \mid x_{\mathsf{r}1}\ \tau_{\mathsf{r}1} \triangleright x_{\mathsf{s}1}} \\
\qquad\qquad \overrightarrow{\textbf{type}\ t_2 = x_{\mathsf{l}2}\ \tau_{\mathsf{l}2} \mid x_{\mathsf{r}2}\ \tau_{\mathsf{r}2} \triangleright x_{\mathsf{s}2}} \\
\qquad\qquad \overrightarrow{\textbf{val}\ x_1{:}\tau_1 = e_1} \\
\qquad\qquad \overrightarrow{\textbf{val}\ x_2{:}\tau_2 = e_2} \\
\qquad\qquad \textbf{in}\ e_{b1}\ ;\ e_{b2}
\end{array}
$$

if $\overrightarrow{t'_a} \cup \overrightarrow{t''_a} \cup \overrightarrow{t'} \cup \overrightarrow{t''} \cup \overrightarrow{x'} \cup \overrightarrow{x''}$ distinct

Fig. B.3. Reduction rules for $\textsc{Unit}_{\mathsf{e}}$

Typed Closure Conversion for Recursively-Defined Functions (Extended Abstract)

Greg Morrisett [1]

Cornell University

Robert Harper [2]

Carnegie Mellon University

Abstract

Much recent work on the compilation of statically typed languages such as ML relies on the propagation of type information from source to object code in order to increase the reliability and maintainabilty of the compiler itself and to improve the efficiency and verifiability of generated code. To achieve this the program transformations performed by a compiler must be cast as type-preserving translations between typed intermediate languages. In earlier work with Minamide we studied one important compiler transformation, closure conversion, for the case of pure simply-typed and polymorphic λ-calculus. Here we extend the treatment of simply-typed closure conversion to account for recursively-defined functions such as are found in ML. We consider three main approaches, one based on a recursive code construct, one based on a self-referential data structure, and one based on recursive types. We discuss their relative advantages and disadvantages, and sketch correctness proofs for these transformations based on the method of logical relations.

[1] This material is based on work supported in part by the AFOSR grant F49620-97-1-0013 and ARPA/RADC grant F30602-96-1-0317. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the authors and do not reflect the views of these agencies.

[2] This research was sponsored by the Advanced Research Projects Agency CSTO under the title "The Fox Project: Advanced Languages for Systems Software", ARPA Order No. C533, issued by ESC/ENS under Contract No. F19628-95-C-0050. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing official policies, either expressed or implied, of the Advanced Research Projects Agency or the U.S. Government.

1 Introduction

Closure conversion is a critical program transformation for higher-order languages that eliminates lexically nested, first-class functions or procedures. In particular, closure conversion translates each function definition f into a *cu-----e* – a data structure consisting of a pointer to closed *c-de* and another data structure which represents the *e--i---ve--* or context of the function. The code abstracts the arguments of f as well as the free variables of f, and the environment provides the values for the free variables of f. Function application is translated to a sequence which invokes the code of the function's closure on the environment of the closure and the arguments. Since the code is closed and separated from the data which it manipulates, it may be defined at the top-level and shared by all closures that are instances of the function. In this respect, closure conversion reifies meta-level constructs (*i.e.*, closures and environments) as object-level constructs (*i.e.*, code and tuples) in the same fashion that conversion into continuation-passing style reifies meta-level continuations as object-level functions.

As an example, the source level expression

$$\begin{aligned}
&\texttt{let } f &&= \lambda x.\lambda y.\lambda z.x + y + z \\
&\quad f' &&= f\,3 \\
&\quad f'' &&= f'\,4 \\
&\texttt{in} \\
&\quad f''\,5 \\
&\texttt{end}
\end{aligned}$$

might be closure-converted to the target language expression

$$\begin{aligned}
&\texttt{let } z_{\text{code}} &&= \lambda[e, z].(\pi_2\, e) + (\pi_1\, e) + z \\
&\quad y_{\text{code}} &&= \lambda[e, y].\langle z_{\text{code}}, \langle y, \pi_1\, e, \rangle\rangle \\
&\quad x_{\text{code}} &&= \lambda[e, x].\langle y_{\text{code}}, \langle x\rangle\rangle \\
&\quad f &&= \langle x_{\text{code}}, \langle\rangle\rangle \\
&\quad f' &&= (\pi_1\, f)\,[\pi_2\, f, 3] \\
&\quad f'' &&= (\pi_1\, f')\,[\pi_2\, f', 4] \\
&\texttt{in} \\
&\quad (\pi_1\, f'')\,[\pi_2\, f'', 5] \\
&\texttt{end}
\end{aligned}$$

The function f becomes a pair of the code x_{code} and an empty environment. The definition of f' invokes f by calling the code of f (x_{code}), passing to it its environment and the argument 3. The code builds a new closure, containing the code y_{code} and the environment consisting of the value bound to x (*i.e.*, 3). Hence, f' becomes bound to the value $\langle y_{\text{code}}, \langle 3 \rangle \rangle$. The definition of f'' invokes f' by calling the code y_{code}, passing to it its environment and the argument 4. The code builds a new closure, containing the code z_{code} and the environment consisting of the values of y (*i.e.*, 4) and x (*i.e.*, 3). Note that the value of x is obtained from the environment e. Hence, f'' becomes bound to the value $\langle z_{\text{code}}, \langle 4, 3 \rangle \rangle$. The body of the `let` invokes this closure on the argument 5. Hence, within the definition of z_{code}, e is bound to $\langle 4, 3 \rangle$ and z is bound to 5. The body of this code computes the value $3 + 4 + 5 = 12$.

In earlier work with Minamide [6] we considered the question of how to perform closure conversion in a typed setting. We sought to relate the type of a program after closure conversion to its type prior to closure conversion. The key observation is that the naïve approach to closure conversion sketched above does not, in general, yield a well-typed term. For instance, consider the following source expression:

$$\texttt{if } c \texttt{ then } \lambda x{:}\mathsf{int}.x + a + b \texttt{ else } \lambda z{:}\mathsf{int}.z$$

Under the typing assumptions c:bool, a:int, and b:int, this expression may be assigned the type $\mathsf{int} \to \mathsf{int}$. A typical closure conversion algorithm yields the following translation:

$$\texttt{if } c \texttt{ then } \langle \lambda[e{:}\langle \mathsf{int} \times \mathsf{int} \rangle, x{:}\mathsf{int}].x + \pi_1\, e + \pi_2\, e, \langle a, b \rangle \rangle$$

$$\texttt{else } \langle \lambda e{:}\langle\,\rangle, z{:}\mathsf{int}.z, \langle\rangle \rangle$$

But this term is not well-typed in a conventional typed λ-calculus since the closure in the `then` clause has type $\langle([\langle \mathsf{int} \times \mathsf{int} \rangle, \mathsf{int}] \to \mathsf{int}) \times \langle \mathsf{int} \times \mathsf{int} \rangle \rangle$ whereas the closure in the `else` clause has type $\langle([\langle\,\rangle, \mathsf{int}] \to \mathsf{int}) \times \langle\,\rangle \rangle$. The issue is that, at the source level, $\mathsf{int} \to \mathsf{int}$ hides the type of the environment, but at the target level, the type of the environment is exposed in the type of the closure.

This problem of representation exposure may be avoided by using an existential type [7] to hide the type of the environment. This ensures that all closures arising from a given source language type have the same type after closure conversion. Specifically, source functions of type $\tau_1 \to \tau_2$ are translated to target closures with type $\exists \alpha.\langle([\alpha, \tau_1] \to \tau_2) \times \alpha \rangle$. This translation is closely related to Pierce and Turner's type system for objects [8] — a function is interpreted as an object with one method (the code) and one instance variable (the environment) using their existential type discipline for simple objects.

The present work is concerned with the extension of our previous work to account for recursively-defined functions. This generalization introduces two

significant complications. First, several different approaches are available, and none dominates the others in all respects. We consider here three translations: one based on recursive code, one based on a self-referential data structure, and one based on recursive types. Second, the correctness proofs given by Minamide, Morrisett, and Harper based on logical relations must be extended to account for recursion. The general idea in each case is to relate the finite approximinants of a recursive function to a corresponding finite approximant of the target code. In the case of recursive code and self-referential data structures, the finite approximants are the finite unrollings of the recursive code or self-referential structure. In the case of recursive types we make use of the syntactic minimal invariant associated with a recursive type [9,5,4] to define the finite approximants of a value.

In the rest of this abstract, we sketch the original simply-typed closure conversion algorithm and the three translations mentioned above for recursive functions.

2 Simply-Typed Closure Conversion

We formalize closure conversion for the simply-typed lambda calculus by defining the syntax, static semantics, and dynamic semantics for a source language ($\lambda^{\rightarrow}$) and a target language ($\lambda^{\exists}$), by giving a type translation $\mathcal{T}[\![\ ]\!]$ from $\lambda^{\rightarrow}$ types to $\lambda^{\exists}$ types, and a term translation from well-formed $\lambda^{\rightarrow}$ terms to $\lambda^{\exists}$ terms.

The syntax for $\lambda^{\rightarrow}$ is:

$$\text{(types) } \tau \ ::= \mathsf{b} \mid \tau_1 \rightarrow \tau_2$$
$$\text{(terms) } e \ ::= \ x \mid \mathsf{c} \mid \lambda x{:}\tau.e \mid e_1\, e_2$$

The language is a conventional simply-typed lambda calculus with a distinguished base type (b) inhabited by a set of constants, over which we use c to range. The static and dynamic semantics (not presented here) is entirely standard.

The syntax of our target language $\lambda^{\exists}$ is:

Types include products (for building both environments and closures), code ($\Rightarrow$) (for the code of closures) and type variables and existentially quantified types (for hiding the type of the environment of a closure). Our target language is *iv--edica-i-e* in that type variables range over quantified as well

as unquantified types.

$$\begin{aligned}
&\text{(types) } \tau ::= t \mid \mathsf{b} \mid \tau_1 \Rightarrow \tau_2 \mid \langle \tau_1, \cdots, \tau_n \rangle \mid \exists t.\tau \\
&\text{(terms) } e ::= x \mid \mathsf{c} \mid \texttt{code } (x{:}\tau).e \mid \texttt{call } e_1(e_2) \mid \\
&\qquad\qquad \langle e_1, \cdots, e_n \rangle \mid \pi_i\, e \mid \\
&\qquad\qquad \texttt{pack}[\tau', e] \texttt{ as } \tau \mid \texttt{let } t, x = \texttt{unpack } e' \texttt{ in } e \mid \\
&\qquad\qquad \texttt{let } x = e' \texttt{ in } e
\end{aligned}$$

Products are introduced and eliminated in the usual fashion. Code types are introduced by `code` terms and eliminated by `call` terms. Existentials are introduced by `pack` terms and eliminated by `unpack` terms. Terms also include a `let` construct which we use to simplify the translation.

The static semantics of $\lambda^{\exists}$ is also standard except for the `code` rule which is similar to the rule for λ-expressions, but requires that the code definition contain no free type variables or value variables. In other words, code definitions are always closed, and can thus always be defined at the top level.

The closure conversion translation is specified by giving a type translation, $\mathcal{T}[\![\ ]\!]$, mapping source types to target types, and a term translate mapping derivable source language typing judgments to target language terms. The type translation is defined as:

$$\begin{aligned}
\mathcal{T}[\![b]\!] &= b \\
\mathcal{T}[\![\tau_1 \rightarrow \tau_2]\!] &= \exists t.\langle(\langle t, \mathcal{T}[\![\tau_1]\!]\rangle \Rightarrow \mathcal{T}[\![\tau_2]\!]), t\rangle
\end{aligned}$$

The arrow type $\tau_1 \rightarrow \tau_2$ is translated to an existentially quantified value. Conceptually, the value is a pair of an abstract type (t) and a product value whose type depends upon t. In this situation, t will abstract the type of the environment of the given closure, thereby avoiding the typing problems mentioned in the introduction. The first component of the product value is code that takes a value of the abstract environment type and an argument of type $\mathcal{T}[\![\tau_1]\!]$, and yields a value of type $\mathcal{T}[\![\tau_2]\!]$. The second component of the product value is a value of the abstract environment type.

The term translation is given in Figure 1. The **var** and **b-I** cases are straightforward. For the **$\rightarrow$-I** case, we first translate the body of the λ-expression, extending Γ with $\{x{:}\tau_1\}$, to yield e'. We then create a closed piece of code that abstracts the free variables of e'. The code has a parameter x_{arg} which is always a pair consisting of the environment and the argument to the function. Upon invocation, the code extracts these parameters and binds them to x_{env} and x respectively. (We assume that x_{arg} and x_{env} are chosen so as to be distinct from all of the variables in the context.) The values for the variables occurring in Γ' are obtained by performing suitable projections on x_{env} and by binding the result to the appropriate variable via `let`. We

$$\textbf{(var)}\ \Gamma \uplus \{x{:}\tau\} \vdash_s x : \tau \rightsquigarrow x \qquad \textbf{(b-I)}\ \Gamma \vdash_s \mathsf{c} : \mathsf{b} \rightsquigarrow \mathsf{c}$$

$$(\rightarrow\textbf{-I})\ \frac{\Gamma \uplus \{x{:}\tau\} \vdash_s e : \tau' \rightsquigarrow e'}{\Gamma \vdash_s \lambda x{:}\tau.e : \tau \rightarrow \tau' \rightsquigarrow \mathsf{pack}[\tau_\Gamma, \langle v_{\text{code}}, v_{\text{env}}\rangle]\ \mathsf{as}\ \mathcal{T}[\![\tau \rightarrow \tau']\!]}$$

$$\begin{aligned}
\text{where}\ \Gamma &= \{x_1{:}\tau_1, \cdots, x_n{:}\tau_n\} \\
\tau_\Gamma &= \langle \mathcal{T}[\![\tau_1]\!], \cdots, \mathcal{T}[\![\tau_n]\!]\rangle \\
v_{\text{env}} &= \langle x_1, \cdots, x_n\rangle \\
v_{\text{code}} &= \mathtt{code}\ (x_{\text{arg}}{:}\langle \tau_\Gamma, \mathcal{T}[\![\tau]\!]\rangle).\ \\
&\qquad \mathtt{let}\ x_{\text{env}} = \pi_1\, x_{\text{arg}}\ \mathtt{in}\ \\
&\qquad \mathtt{let}\ x = \pi_2\, x_{\text{arg}}\ \mathtt{in}\ \\
&\qquad \mathtt{let}\ x_1 = \pi_1\, x_{\text{env}}\ \mathtt{in}\ \\
&\qquad \vdots\ \\
&\qquad \mathtt{let}\ x_n = \pi_n\, x_{\text{env}}\ \mathtt{in}\ \\
&\qquad e'
\end{aligned}$$

$$(\rightarrow\textbf{-E})\ \frac{\Gamma \vdash_s e_1 : \tau_2 \rightarrow \tau \rightsquigarrow e_1' \qquad \Gamma \vdash_s e_2 : \tau_2 \rightsquigarrow e_2'}{\Gamma \vdash_s e_1\, e_2 : \tau \rightsquigarrow} \quad (x \notin D\text{-}v(\Gamma))$$

$$\mathtt{let}\ t, x = \mathtt{unpack}\ e_1'\ \mathtt{in}\ \mathtt{call}\ (\pi_1\, x)(\langle \pi_2\, x, e_2'\rangle)$$

Fig. 1. Closure Conversion for $\lambda^{\rightarrow}$

create the environment by building a tuple $\langle x_1, \cdots, x_n\rangle$ containing the values of those variables in Γ. The code and the environment tuple are placed in a pair, and the data structure is packed in order to abstract the type of the environment.

For the $\rightarrow$-**E** case, we translate the function and argument to yield e_1' and e_2' respectively. We then unpack e_1' and bind the abstract type to the type variable t and the contents of the closure to x. We then project the code from x followed by the environment, and call the code passing it the environment and argument e_2'.

3 Recursive Closure Conversion

In this section, we show how to extend the simply-typed closure conversion to deal with recursive functions. Interestingly, there are many possible translations, each with different tradeoffs in terms of efficiency and/or complexity of the resulting type system and semantics for the target language. A fascinating aspect is that all of the translations presented here have a direct correspon-

dence to type encodings used for various kinds of object-oriented languages that support a notion of "self".

Our source language is the same as $\lambda^{\rightarrow}$, but we extend values with fix-expressions on abstractions:

$$\text{(terms) } e ::= \cdots \mid \texttt{fix } x(x_1{:}\tau_1){:}\tau_2.e$$

Here, both x and x_1 are bound within e. Though our source language only supports single recursion, it is straightforward but tedious to extend the presentation in the following sections to mutual recursion.

3.1 The Fix-C-de T-a--ua-i--

In this section, we use recursive *c-de* definitions in the closure conversion translation. The translation is entirely straightforward in that the type translation is the same as for simply-typed closure conversion and no recursive (*i.e.*, cyclic) data structures are required. Hence, the "fix-code" translation is suitable for use with a reference-counting garbage collector. Furthermore, the fix-code translation supports easy optimization of known functions as with the original translation. However, the implementation leads to unecessary duplication of closures. In essence, a copy of the closure is re-created each time the function is invoked.

The only change to the target language is the addition of a `fixcode` construct:

$$\text{(terms) } e ::= \cdots \mid \texttt{fixcode } x(x{:}\tau_1){:}\tau_2.e$$

The code may only refer to its arguments or itself. The dynamic semantics "unrolls" the code at the point where it is called, just as fix is normally unrolled.

Closure conversion from the source to the target is straightforward. The type translation remains the same, as does the translation of application and

λ. The only addition is the translation rule for `fix`:

$$\textbf{(fix-1)}\quad \frac{\Gamma \uplus \{x{:}\tau' \to \tau, x'{:}\tau'\} \vdash_s e : \tau \rightsquigarrow e'}{\begin{array}{l}\Gamma \vdash_s \texttt{fix}\ x(x'{:}\tau'){:}\tau.e : \tau' \to \tau \rightsquigarrow \\ \quad \texttt{pack}[\tau_\Gamma, \langle v_{\text{code}}, v_{\text{env}}\rangle]\ \texttt{as}\ \mathcal{T}[\![\tau \to \tau']\!]\end{array}}$$

where

$$\begin{aligned}
\Gamma &= \{x_1{:}\tau_1, \cdots, x_n{:}\tau_n\} \\
\tau_\Gamma &= \langle \mathcal{T}[\![\tau_1]\!], \cdots, \mathcal{T}[\![\tau_n]\!]\rangle \\
v_{\text{env}} &= \langle x_1, \cdots, x_n\rangle \\
v_{\text{code}} &= \texttt{fixcode}\ x_c(x_{\text{arg}}{:}\langle \tau_\Gamma, \mathcal{T}[\![\tau]\!]\rangle). \\
&\qquad \texttt{let}\ x_{\text{env}} = \pi_1\, x_{\text{arg}}\ \texttt{in} \\
&\qquad \texttt{let}\ x' = \pi_2\, x_{\text{arg}}\ \texttt{in} \\
&\qquad \texttt{let}\ x = \texttt{pack}[\tau_\Gamma, \langle x_c, x_{\text{env}}\rangle]\ \texttt{as}\ \mathcal{T}[\![\tau \to \tau']\!]\ \texttt{in} \\
&\qquad \texttt{let}\ x_1 = \pi_1\, x_{\text{env}}\ \texttt{in} \\
&\qquad \vdots \\
&\qquad \texttt{let}\ x_n = \pi_n\, x_{\text{env}}\ \texttt{in} \\
&\qquad e'
\end{aligned}$$

The environment consists of the free variables of the body of the function except for x (the function itself) and x' (the argument). The closure for x is constructed as before by pairing the code with the environment. However, e' is obtained from e in a context where x must be available as a closure – not just code, as the closure could "escape" (*i.e.*, be passed as an argument to another function.) Hence, once we enter the code, we have to construct a "new" copy of the closure for use within the body of the code. Therefore, we create a new closure from x_c and x_{env} and bind it to x. Of course, if x is only called within the body e, then the reductions can eliminate the unnecessary construction of this value. But in general, we have to allocate a new closure pair each time around the loop which in practice actually is quite costly. The other translations attempt to address this by constructing the closure exactly once.

Note that the situation is even worse for mutually recursive functions. Suppose we've defined $f_1, \cdots, f_n$ via a letrec. In general, we have to construct *each* of the closures for $f_1, \cdots, f_n$ out of the code and shared environment each time one of these functions is called.

3.2 *The Fix-Pact T-a--ua-i--*

In our second translation, the key idea is that instead of adding recursive code to the target language, we add a recursive `pack` construct that allows us to define a closure data structure in terms of itself. The well-known trick is to build a circular data structure to represent `fix` closures where the environment for the closure contains the closure itself. In a sense, the previous translation is building this circular data structure lazily each time the closure is invoked. Our goal with this translation is to build the circular data structure once, avoiding the overhead of allocating a new copy each time the code is invoked.

To accomplish this, we add to the original target language the following term constructs:

$$\text{(terms) } e ::= \cdots \mid \texttt{fixpack } x.[\tau', v] \texttt{ as } \tau$$

The new construct allows one to define a recursive package of existential type. Like a recursive function, the variable x is bound within the body v of the term.

With the new target language construct, we define the recursive closure conversion translation as follows: The type translation is the same as for the first two translations. The term translation is also the same for application and λ, but the translation for `fix` is:

$$\textbf{(fix-2)}\quad \frac{\Gamma \uplus \{x{:}\tau \to \tau', x'{:}\tau\} \vdash_s e : \tau' \rightsquigarrow e'}{\begin{array}{l}\Gamma \vdash_s \texttt{fix } x(x'{:}\tau){:}\tau'.e : \tau \to \tau' \rightsquigarrow \\ \quad \texttt{fixpack } x.[\tau_{\text{env}}, \langle v_{\text{code}}, v_{\text{env}}\rangle] \texttt{ as } \mathcal{T}[\![\tau \to \tau']\!]\end{array}}$$

$$\begin{array}{rl}\text{where } \Gamma & = \{y_1{:}\tau_1, \cdots, y_n{:}\tau_n\} \\ \tau_{\text{env}} & = \langle \mathcal{T}[\![\tau \to \tau']\!], \mathcal{T}[\![\tau_1]\!], \cdots, \mathcal{T}[\![\tau_n]\!]\rangle \\ v_{\text{env}} & = \langle x, y_1, \cdots, y_n\rangle \\ v_{\text{code}} & = \texttt{code } (x_{\text{arg}} : \langle \tau_{\text{env}}, \mathcal{T}[\![\tau]\!]\rangle) : \mathcal{T}[\![\tau']\!]. \\ & \qquad \texttt{let } x_{\text{env}} = \pi_1\, x_{\text{arg}} \texttt{ in} \\ & \qquad \texttt{let } x' = \pi_2\, x_{\text{arg}} \texttt{ in} \\ & \qquad \texttt{let } x = \pi_1\, x_{\text{env}} \texttt{ in} \\ & \qquad \texttt{let } y_1 = \pi_2\, x_{\text{env}} \texttt{ in} \\ & \qquad \vdots \\ & \qquad \texttt{let } y_n = \pi_{n+1}\, x_{\text{env}} \texttt{ in} \\ & \qquad e'\end{array}$$

3.3 *The Fix-Ty-e --a--ua-i--*

The previous translations used an environment-passing strategy where the code of a closure is passed the environment as an extra argument. In this translation, instead of passing the environment, we pass the closure itself as an extra argument to the code of the closure. Right away, it's obvious that the closure must contain a recursive type because the code is contained in the closure, but the code takes the closure as an argument. Hence, a possible type translation is given by:

$$\mathcal{T}[\![b]\!] = b$$

$$\mathcal{T}[\![\tau_1 \to \tau_2]\!] = \exists t_{\text{env}}.\mu t_{\text{cl}}.\langle (t_{\text{cl}}, \mathcal{T}[\![\tau_1]\!]) \Rightarrow \mathcal{T}[\![\tau_2]\!], t_{\text{env}} \rangle$$

Given this, the previous translations for application are correct, modulo the insertion of an "unroll" (assuming we want the isomorphism between the rolled and unrolled versions of a recursive type to be made explicit):

$$\textbf{(app-3)} \quad \frac{\Gamma \vdash_s e_1 : \tau_2 \to \tau \rightsquigarrow e_1' \qquad \Gamma \vdash_s e_2 : \tau_2 \rightsquigarrow e_2'}{\Gamma \vdash_s e_1\, e_2 : \tau \rightsquigarrow}$$

$$\texttt{let } t, x = \texttt{unpack } e_1' \texttt{ in call } (\pi_1\,(\texttt{unroll}(x)))(x, e_2')$$

The term translation for fix becomes:

$$\textbf{(fix-3)} \quad \frac{\Gamma \uplus \{x{:}\tau_1 \to \tau_2, x'{:}\tau_1\} \vdash_s e : \tau_2 \rightsquigarrow e'}{\Gamma \vdash_s \texttt{fix } x(x'{:}\tau_1){:}\tau_2.e : \tau_1 \to \tau_2 \rightsquigarrow}$$

$$\texttt{pack}[\tau_\Gamma, \texttt{roll}(\langle v_{\text{code}}, v_{\text{env}} \rangle{:}\tau_{\text{cl}})] \texttt{ as } \mathcal{T}[\![\tau_1 \to \tau_2]\!]$$

where

$$\begin{aligned}
\Gamma &= \{x_1{:}\tau_1, \cdots, x_n{:}\tau_n\} \\
\tau_\Gamma &= \langle \mathcal{T}[\![\tau_1]\!], \cdots, \mathcal{T}[\![\tau_n]\!] \rangle \\
\tau_{\text{cl}} &= \mu t_{\text{cl}}.\langle (t_{\text{cl}}, \mathcal{T}[\![\tau_1]\!]) \Rightarrow \mathcal{T}[\![\tau_2]\!], \tau_\Gamma \rangle \\
v_{\text{env}} &= \langle x_1, \cdots, x_n \rangle \\
v_{\text{code}} &= \texttt{code } x_c(x_{\text{cl}}{:}\tau_{\text{cl}}, x{:}\mathcal{T}[\![\tau_1]\!]) : \mathcal{T}[\![\tau_2]\!]. \\
&\qquad \texttt{let } x = \texttt{pack}[\tau_\Gamma, x_{\text{cl}}] \texttt{ as } \mathcal{T}[\![\tau_1 \to \tau_2]\!] \texttt{ in} \\
&\qquad \texttt{let } x_{\text{env}} = \pi_2\,(\texttt{unroll}(x_{\text{cl}})) \texttt{ in} \\
&\qquad \texttt{let } x_1 = \pi_1\, x_{\text{env}} \texttt{ in} \\
&\qquad \vdots \\
&\qquad \texttt{let } x_n = \pi_n\, x_{\text{env}} \texttt{ in} \\
&\qquad e'
\end{aligned}$$

This translation has all of the (operational) advantages of the fixpack approach. In particular, the closure is only created once and not each time

around the loop as in the fixcode case.

There are a couple of relatively minor reasons why this translation might be preferred over the fixpack translation: In particular, since the code, not the caller, extracts the environment from the closure, a slight optimization is possible when the environment is empty, as the code can avoid projecting the environment. In contrast, for all of the other translations, the caller extracts the environment. Since the caller cannot in general know the type of the environment (*i.e.*, whether it is unit), the caller cannot know whether the environment is actually needed or not. Furthermore, with suitable support in the target language, the environment can be "flattened" into the closure tuple. That is, instead of $\langle v_{\mathsf{code}}, \langle v_1, \cdots, v_n \rangle \rangle$, we can represent the closure as $\langle v_{\mathsf{code}}, v_1, \cdots, v_n \rangle$, thereby avoiding extra allocation and indirection. It is worth remarking that this is the closure-passing style that Appel and Jim proposed in an untyped setting [2], and was used until recently in the SML/NJ compiler [3].

Closure-passing does have its drawbacks: it requires recursive types (not just monotonic types) which seriously complicates the semantics of the target language, and to reap the allocation benefits, requires some rather *ad h-c* data structures (*e.g.*, pointers into the middle of tuples [1].)

References

[1] A. W. Appel. *Compiling with Continuations.* Cambridge University Press, 1992.

[2] A. W. Appel and T. Jim. Continuation-passing, closure-passing style. In *Sixteenth ACM Symposium on Principles of Programming Languages*, pages 293–302, Austin, TX, January 1989.

[3] A. W. Appel and D. B. MacQueen. Standard ML of New Jersey. In J. Maluszynski and M. Wirsing, editors, *Third Int'l Symp. on Prog. Lang. Implementation and Logic Programming*, pages 1–13, New York, August 1991. Springer-Verlag.

[4] L. Birkedal and R. Harper. Relational interpretations of recursive types in an operational setting (summary). In *Theoretical Aspects of Computer Science*, Sendai, Japan, September 1997. (To appear.).

[5] I. A. Mason, S. F. Smith, and C. L. Talcott. From operational semantics to domain theory. *Information and Computation*, 128(1):26–47, 1996.

[6] Y. Minamide, G. Morrisett, and R. Harper. Typed closure conversion. In *Twenty-Third ACM Symposium on Principles of Programming Languages*, pages 271–283, St. Petersburg, FL, January 1996.

[7] J. C. Mitchell and G. Plotkin. Abstract types have existential type. *ACM Transactions on Programming Languages and Systems*, 10(3):470–502, 1988.

[8] B. C. Pierce and D. N. Turner. Object-oriented programming without recursive types. In *Twentieth ACM Symposium on Principles of Programming Languages*, Charleston, SC, Jan. 1993.

[9] A. M. Pitts. Relational properties of domains. *Information and Computation*, 127(2):66–90, June 1996.